

一：NE 框架

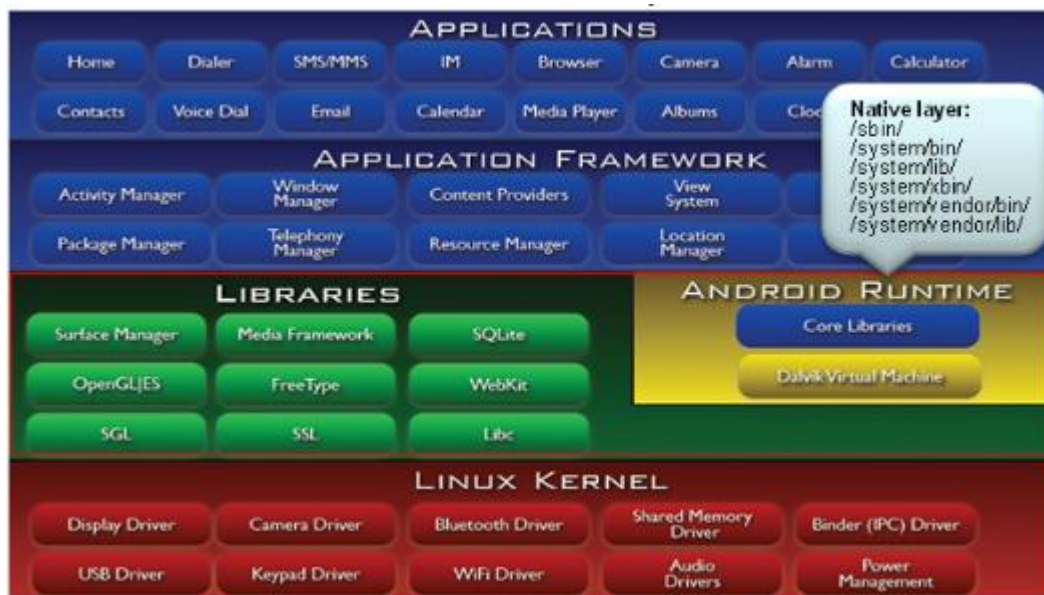
1. Native application

本地应用程序是指可以直接运行在操作系统上，并且处理器直接执行机器码的程序。

比如 windows 上的各种*.exe 的程序，而 linux 上的是各种 bin 程序。

在 Android 上，OS 是 linux，因此各种 bin 程序就是所谓 native application 了，比如/system/bin 目录下的所有文件。这些应用程序都是由 GCC(c/c++) 编译生成。

在 Android 软件架构里，这些应用程序组成了 native layer：



2. Native Exception

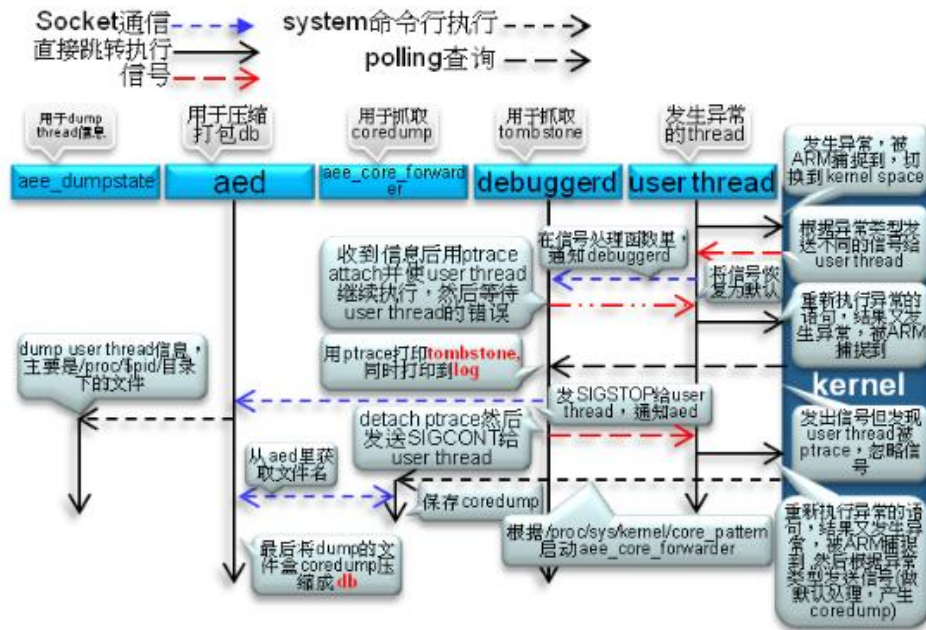
native layer 里的应用程序崩溃统称为 Native Exception，比如空指针，非法指针，程序跑飞，内存踩坏等，好比像 windows 下，程序崩溃弹出某某地址不能为 read/write。

3. 总流程图

原始的 linux，对于用户进程崩溃之后，处理方式有 2 种：直接终止进程；输出 coredump 再终止进程。

而在 Android，为了方便调试，在收到崩溃信号后，会先输出 tombstone，然后在根据设置是否抓取 coredump，最后再终止进程。而我司在这之上还会将 coredump 及其他关键信息打包。

以下是完整的 NE 处理流程图：



4. 例子

我们以 1 个 NE 的例子来将流程走一遍。

首先写 test.c:

```
void d()
{
    char *p = NULL;
    *p = 0x323; /* 这里让程序崩溃 */
}

void c()
{
    int var4 = 6;
}

void b()
{
    int var3 = 3;
    c();
    d();
}

void a()
{
    int var1, var2;
    var1 = 1;
}

int main(int argc, char *argv[])
{
    int var0 = 4;
    a();
    b();
    return 0;
}
```

然后编写 Android.mk:

```
LOCAL_PATH:=$(call my-dir)

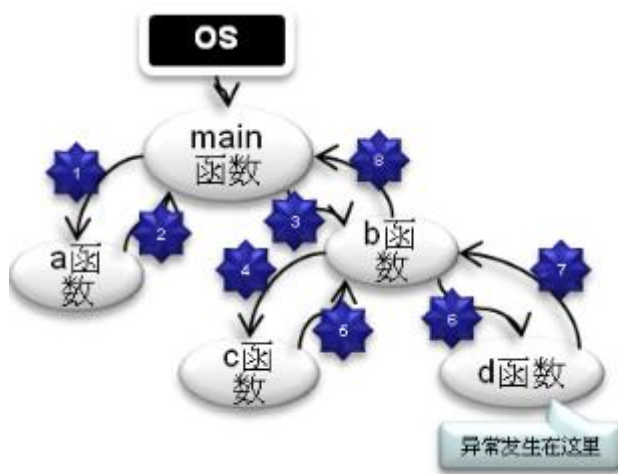
include $(CLEAR_VARS)

LOCAL_SRC_FILES:=test.c

LOCAL_MODULE:=test
```

```
include $(BUILD_EXECUTABLE)
```

编译后将 test 推送到手机端，然后执行它，其中走过的函数步骤如下：



当走到 d() 函数（也就是第 6 步之后）时必然产生 1 个 NE。

二：用户空间布局

1. 布局

native 程序是运行在 linux 之上的,因此程序如何被 linux 加载起来,如何发生/捕获/处理异常都需要了解的清清楚楚,这样才能系统的分析 NE。我们先从 user space layout 入手。

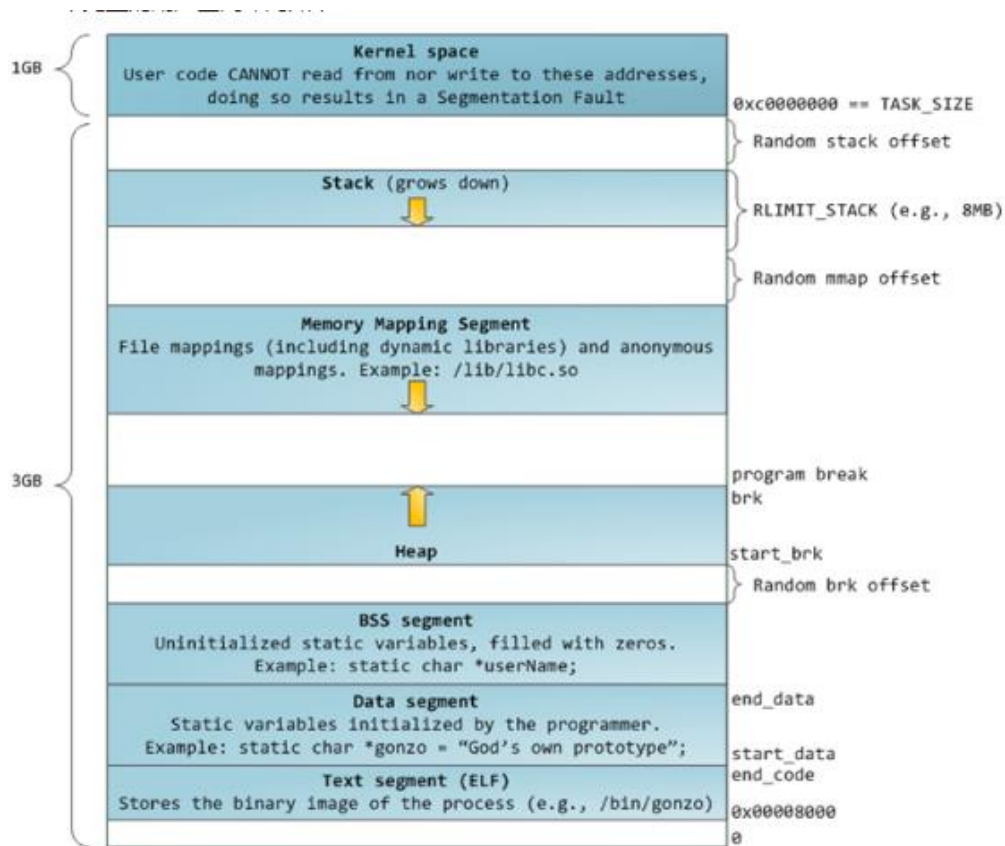
对于 32 位 ARM 来说，最大能访问的空间是 4G，其中 kernel 占用了最高 1G 空间，是所有进程都共享的，剩下的低 3G 的空间是每个应用程序独有和互不干扰，这样的布局是由 MMU 帮忙实现的。

应用程序内容:

主要可以分为 3 个部分：只读的代码段和数据段，可读写的数据段，bss 未初始化数据段。

程序的 3 个部分会被加载到用户空间,同时会分配一个栈给程序,如果程序运行过程中调用 `malloc()` 等堆内存分配函数,则会将堆映射到用户空间;如果调用 `mmap()`,则会将文件映射到用户空间。

一个完整的用户空间布局如下：



2. 解剖

(1). Text segment

也就是程序里的代码段和只读数据段，在 ICS 及以前版本是固定加载在 `0x8000` 的位置上，JB 及之后为了安全的考量，则是随机加载在 `0x40000000` 的位置上（这个地址之上属于 mmap 空间）

【问题】加载的位置定义在哪里？

【解答】在 gcc 连接脚本里指定，ICS 及之前的版本在 `alps/build/core/armelf.x` 里有指定 `__executable_start = 0x8000`，JB 及之后的版本则取消该文件了。

(2). Data segment/BSS segment

在 Text segment 之上就是 data 和 bss 了，这 3 段是由系统帮忙加载，系统是根据应用程序文件里的信息加载的，而应用程序格式在 linux 上是 ELF（如同 windows 的 PE 格式）。

(3). Stack

这个栈是系统分配的初始栈，里面存放环境变量，辅助向量，参数等信息。

这个栈的大小是受系统限制的，参数是：RLIMIT_STACK，可以通过 `ulimit -s` 查看（单位是 KB）：

```
#ulimit -s
8192
```

栈底为 `0xc0000000` 随机偏移 8M（random stack offset）的位置，随机的目的也是为了安全考量。对应内核代码如下：

```
static unsigned long randomize_stack_top(unsigned long stack_top)
{
    unsigned int random_variable = 0;

    if ((current->flags & PF_RANDOMIZE) &&
        !(current->personality & ADDR_NO_RANDOMIZE)) {
        random_variable = get_random_int() & STACK_RND_MASK; /* 8MB of VA */
    }
}
```

【问题】如果在程序里创建其他线程，栈是如何配置的？是否可以自动增长？

【解答】线程都是通过 pthread 库创建，其中 pthread_create() 函数就是用于创建线程的，你可以指定栈的空间大小，由该函数帮你申请（通过 mmap），这个无法自动增长。

也可以自己 malloc/mmap 等方法申请，再传给该函数，如果是 mmap，可以设定是否自动增长。

(4). Heap

应用程序从系统要内存只能通过系统调用：mmap/brk，这里拿到的内存都是按页对齐的（32 位 ARM 是 4K）。

brk 拿到的内存有个特性，向上增长，顶部可以自由伸缩（也就是可以扩展/释放堆）。

c 常用的 malloc 首先通过 brk 调用拿到内存（如何 brk 失败则通过 mmap），然后在经过管理给你指定大小的内存。

start_brk 为 BSS 段随机偏移 32M（random brk offset）的位置，随机的目的也是为了安全考量。对应内核代码如下：

```
unsigned long arch_randomize_brk(struct mm_struct *mm)
{
    unsigned long range_end = mm->brk + 0x02000000;
```

(5). Memory Mapping Segment

通过 mmap 系统调用映射进来的，都会落在这个区域（除非指定了其他地址）。包括动态库和文件。

该区域存在 2 个增长方向：向上增长和向下增长。

向上增长：起始地址是 TASK_UNMAPPED_BASE（0x40000000）。

3. 地址随机化

如上一章节讲到，栈/堆/共享库的位置都有一个随机量，目的是增加攻击者预测目的地址的难度，防止攻击者直接定位攻击代码位置，达到阻止溢出攻击的目的。

该技术简称 ASLR（Address space layout randomization）：一种针对缓冲区溢出的安全保护技术。

【问题】如何关闭 ASLR？

【解答】通过设置 kernel.randomize_va_space 内核参数来设置内存地址随机化的行为：

```
#echo 0 >/proc/sys/kernel/randomize_va_space
```

目前 randomize_va_space 的值有三种，分别是[0, 1, 2]

0：关闭进程地址空间随机化

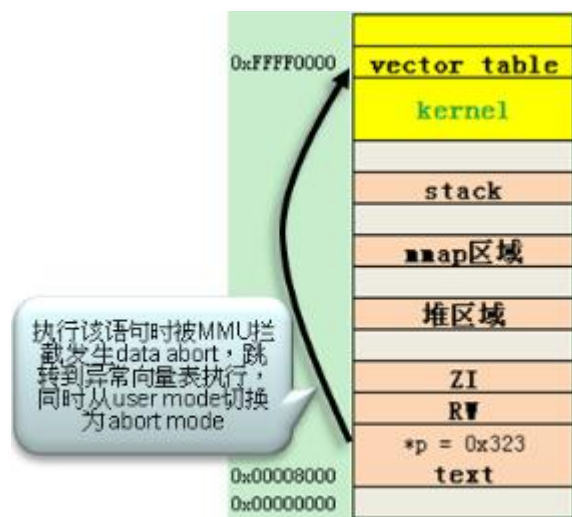
1：将 mmap 的基址，stack 和 vdso 页面随机化

2：在 1 的基础上增加堆(heap)的随机化

三：流程-异常处理

1. 用户空间产生异常

d()函数里对空指针赋值，由于 0 地址属于非法地址，访问时会被 MMU（内存保护单元）拦截，并发送 abort 信号给 CPU：



ARM CPU 有 1 张异常向量表响应对应的异常（可以存放在 0x00000000 或 0xFFFF0000 处，目前配置为 0xFFFF0000），访问 0 地址对应的是 data abort，如果程序跑飞则对应的是 prefetch abort:

Exception type	Mode
Reset	Supervisor
Undefined instructions	Undefined
Software interrupt (SWI)	Supervisor
Prefetch Abort (instruction fetch memory abort)	Abort
<u>Data Abort (data access memory abort)</u>	Abort
IRQ (interrupt)	IRQ
FIQ (fast interrupt)	FIQ

此时从用户态切换到了内核态，在内核态，会将当前的上下文保存并记录异常的地址和类型，内核处理完成后在返回用户态。

以下是对应的异常向量表的汇编代码（对应的代码：alps/kernel/arch/arm/kernel/entry-armv.s）:

异常向量表

lr/line	mnemonic	comment
FFFF0000	svc	0x9F0000
FFFF0004	b	0xFFFF0300
FFFF0008	ldr	pc,0xFFFF0420
FFFF000C	b	0xFFFF0300
FFFF0010	b	0xFFFF0280
FFFF0014	b	0xFFFF0404
FFFF0018	b	0xFFFF0200
FFFF001C	mrs	r12,spsr
FFFF0020	sub	r11,r14,#0x4
FFFF0024	subs	r10,#0x1
FFFF0028	bne	0xFFFF00D0
FFFF002C	push	{r11-r12,r14}
FFFF0030	sub	r13,r13,#0x1C
FFFF0034	stm	r13,{r0-r14}^
FFFF0038	nop	

addr/line	mnemonic	comment
NSD:FFFF0280	sub	r14,r14,#0x8
NSD:FFFF0284	stm	r13,{r0,r14}
NSD:FFFF0288	mrs	r14,spsr
NSD:FFFF028C	str	r14,[r13,#0x8]
NSD:FFFF0290	mrs	r0,cpsr
NSD:FFFF0294	eor	r0,r0,#0x4
NSD:FFFF0298	msr	spsr_cxsf,r0
NSD:FFFF029C	and	r14,r14,#0x0F
NSD:FFFF02A0	cpy	r0,r13
NSD:FFFF02A4	ldr	r14,[pc,#+r14,ls1 #0x2]
NSD:FFFF02A8	movs	pc,r14

最终跳转到kernel的异常处理函数，读取发生异常的地址，异常的类型，送出信号

2. 内核异常处理流程

1 张图总结内核异常处理流程:

Table 3-5 Priority encoding of fault status

Priority	Sources		FS [3:0]	Domain [3:0]	FAR
Highest	Terminal Exception		0b0010	Invalid	IMPLEMENTATION DEFINED
	Vector Exception		0b0000	Invalid	Valid
	Alignment		0b00x1	Invalid	Valid
	External Abort on Translation	First level	0b1100	Invalid	Valid
		Second level	0b1110	Valid	Valid
	Translation	Section	0b0101	Invalid	Valid
		Page	0b0111	Valid	Valid
Lowest	Domain	Section	0b1001	Valid	Valid
		Page	0b1011	Valid	Valid
	Permission	Section	0b1101	Valid	Valid
		Page	0b1111	Valid	Valid
	External Abort on Linefetch	Section	0b0100	Valid	Valid
		Page	0b0110	Valid	Valid
	External Abort on Non-linefetch	Section	0b1000	Valid	Valid
		Page	0b1010	Valid	Valid

dataabort无法处理异常后抛出的信号

```
static struct FSR_INFO fsr_info[] = {
    { do_bad,          SIGSEGV, 0,          "vector exception"
    { do_bad,          SIGBUS,  BUS_ADRALN, "alignment exception"
    { do_bad,          SIGKILL, 0,          "terminal exception"
    { do_bad,          SIGBUS,  BUS_ADRALN, "alignment exception"
    { do_bad,          SIGBUS,  0,          "external abort on linefetch"
    { do_translation_fault, SIGSEGV, SEGV_MAPERR, "section translation fault"
    { do_bad,          SIGBUS,  0,          "external abort on linefetch"
    { do_page_fault,    SIGSEGV, SEGV_MAPERR, "page translation fault"
    { do_bad,          SIGBUS,  0,          "external abort on non-linefetch"
    { do_bad,          SIGSEGV, SEGV_ACCERR, "section domain fault"
    { do_bad,          SIGBUS,  0,          "external abort on non-linefetch"
    { do_bad,          SIGSEGV, SEGV_ACCERR, "page domain fault"
    { do_bad,          SIGBUS,  0,          "external abort on translation"
    { do_sect_fault,    SIGSEGV, SEGV_ACCERR, "section permission fault"
    { do_bad,          SIGBUS,  0,          "external abort on translation"
    { do_page_fault,    SIGSEGV, SEGV_ACCERR, "page permission fault"

```

如果是缺页异常，则会调用 `do_page_fault()`/`do_translation_fault()`，最后返回。但是 0 地址不在进程空间内，因此会转调 `arm_notify_die()`，最后发出信号给自己（注意：是自己发出信号给自己!!!）

`_send_signal()` 函数有添加 `printk`：

```
static int __send_signal(int sig, struct siginfo *info, struct task_struct *t, int group, int from)
{
    printk(KERN_DEBUG "[%d:%s] sig %d to [%d:%s]\n", current->pid, current->comm, sig, t->pid, t->comm);

```

这时在 kernel log 看到对应的 log：

```
<7>[ 695.316153] (0) [3518:putmethod.] sig 11 to [3518:putmethod.]
```

【TIPS】如果发生 NE，可以通过 kernel log 搜索 sig 11/7 等几个可以导致进程崩溃的关键字判断。

进程在收到信号时会做出相应的处理，具体需要查看内核信号章节。

到这里，我们在内核绕了 1 圈又要返回用户空间了。

4. 信号捕获

大家会想，`test.c` 里没有任何信号处理相关的代码，如何能处理信号？

这就需要程序被系统加载说起了。程序分为 2 种，1 种为静态链接的程序，另一种为动态链接的程序。


静态链接：由链接器在链接时将库的内容加入到可执行程序中的做法。最大缺点是生成的可执行文件太大，需要更多

的系统资源，在装入内存时也会消耗更多的时间。

动态链接：将独立的模块先编译成动态库（比如 libc.so/libutils.so 等），程序运行有需要它们时才加载。最大缺点是可执行程序依赖分别存储的库文件才能正确执行。如果库文件被删除了，移动了，重命名了或者被替换为不兼容的版本了，那么可执行程序就可能工作不正常。

在 Android 里，大部分都是动态链接，而只有 init 等少部分是静态链接。因此我们的 test 也是动态链接程序，动态链接程序是需要链接器才能跑起来，linker 就是 Android 的链接器。

我们看下 /proc/\$pid/maps 或 db 里的 PROCESS_MAPS：



00008000-0001a000	r-xp	00000000	b3:04	116	/system/bin/atcid
0001a000-0001b000	rw-p	00012000	b3:04	116	/system/bin/atcid
0001b000-00024000	rw-p	00000000	00:00	0	
01691000-01698000	rw-p	00000000	00:00	0	[heap]
40081000-4008e000	r--s	00000000	00:0b	2309	/dev/__properties__ (deleted)
40097000-400fc000	r-xp	00000000	b3:04	698	/system/lib/libc.so
400fc000-400ff000	rw-p	00065000	b3:04	698	/system/lib/libc.so
400ff000-4010a000	rw-p	00000000	00:00	0	
b0001000-b000a000	r-xp	00001000	b3:04	188	/system/bin/linker
b000a000-b000b000	rw-p	0000a000	b3:04	188	/system/bin/linker
b000b000-b0018000	rw-p	00000000	00:00	0	
beb29000-beb4a000	rw-p	00000000	00:00	0	[stack]
ffff0000-ffff1000	r-xp	00000000	00:00	0	[vectors]

linker 也在程序的进程空间内。当内核将应用程序加载起来后，并不是先跑应用程序代码，而是先跑 linker。linker 负责将应用程序所需的库加载到进程空间内，之后才跑应用程序代码。

【TIPS】kernel 加载完应用程序/动态链接器后并不帮忙加载所需的动态库，这些工作只能由动态链接器完成。动态链接器可以使用 2 种方式加载动态库：立即模式：将所需的动态库一次性全部加载，这样可以提高性能但延长加载时间。懒惰模式：程序需要的时候再加载，前期加载时间快但是如果程序在执行关键的代码，动态链接会影响性能。

linker 执行期间还做了一件事：注册信号！，具体的函数调用流程如下：

__linker_init() -> __linker_init_post_relocation() -> debuggerd_init()


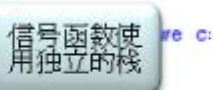
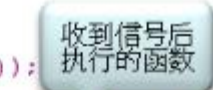
```
void debuggerd_init() {
    struct sigaction action;
    memset(&action, 0, sizeof(action));
    sigemptyset(&action.sa_mask);
    action.sa_sigaction = debuggerd_signal_handler;
    action.sa_flags = SA_RESTART | SA_SIGINFO;

    // Use the alternate signal stack if we can
    action.sa_flags |= SA_ONSTACK;

    sigaction(SIGABRT, &action, NULL);
    sigaction(SIGBUS, &action, NULL);
    sigaction(SIGFPE, &action, NULL);
    sigaction(SIGILL, &action, NULL);
    sigaction(SIGPIPE, &action, NULL);
    sigaction(SIGSEGV, &action, NULL);

    sigaction(SIGSTKFLT, &action, NULL);

    sigaction(SIGTRAP, &action, NULL);
} ? end debuggerd_init ?
```



信号的知识，请查看：[编程篇：linux c 编程的信号章节](#)。

5. 信号处理

目前会产生 native exception (NE) 的几个信号需要特别掌握产生的原因，这样才能进一步分析问题所在。

内核发送信号过来后会执行 debuggerd_init() 里注册的函数 debugger_signal_handler()，该函数会打印基本信息到 main log：

logSignalSummary() 函数会输出基本异常信息：



```

470 470 D InCallScreen: [okToDialDTMF] state: DISCON
470 470 D InCallControlState: InCallControlState:
470 7915 F libc : Fatal signal 11 (SIGSEGV) at 0x0000003c (code=1)
470 470 D InCallControlState: manageConferenceVisible: false
470 470 D InCallControlState: manageConferenceEnabled: false
7515 7515 D AEE/AED : $===AEE===AEE===AEE===

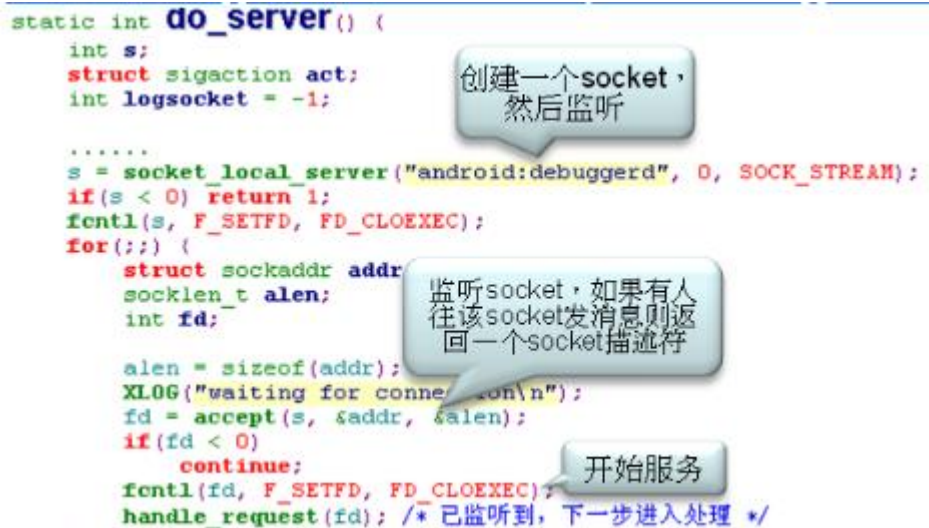
```

然后连接 debuggerd 的 socket, 将 pid 等信息发送给 debuggerd, 请它帮忙后续的处理。之后将对应的信号恢复为默认, 等待 debuggerd 处理完后抓取 coredump。

四: 流程-debuggerd

1. debuggerd 服务

init.rc 会将 debuggerd 拉起来, 具体代码请查看 system/core/debuggerd/目录。debuggerd 起来后会创建 1 个 socket, 然后监听, 等待别人通过 socket 请求服务, 服务可以是生成 tombstone 或调用栈:



```

static int do_server() {
    int s;
    struct sigaction act;
    int logsocket = -1;

    .....
    s = socket_local_server("android:debuggerd", 0, SOCK_STREAM);
    if(s < 0) return 1;
    fcntl(s, F_SETFD, FD_CLOEXEC);
    for(;;) {
        struct sockaddr addr;
        socklen_t alen;
        int fd;

        alen = sizeof(addr);
        XLOG("waiting for connection\n");
        fd = accept(s, &addr, &alen);
        if(fd < 0)
            continue;
        fcntl(fd, F_SETFD, FD_CLOEXEC);
        handle_request(fd); /* 已监听到, 下一步进入处理 */
    }
}

```

【TIPS】当然你也可以单独调用 debuggerd

debuggerd -b \$tid. 抓取指定 tid 的调用栈。

debuggerd \$tid. 抓取指定 tid 的 tombstone。

2. ptrace attach/detach

通过 socket 拿到 tid 等资料后, 使用 ptrace attach 上目标 tid, 之后就可以通过 ptrace 访问目标进程空间, 然后打印一些 NE 相关的寄存器/调用栈等信息。

(1). ptrace attach

ptrace attach 会发送 sig 19 给对应的进程。在这里, 我们将进程内所有线程都 attach 上, 防止有线程提前退出。从 log 可以看到这个动作:



(2). ptrace cont

attach 之后还不能直接访问目标进程，因为目标进程还处于信号处理函数里面，我们需要让它恢复到异常现场，因此需要用 ptrace cont 让其继续执行。

(3). waitpid

程序接着往下跑必然会再次发生异常（如果是 SIGABRT, SIGFPE, SIGPIPE, SIGSTKFLT，则会在信号处理函数补刀，重发一次信号），kernel 会再次发出信号，只不过由于进程被 ptrace 了，信号会发送给 debuggerd。

【TIPS】到这里，目标进程已经收到 2 次同样的信号了！！

(4). tombstone

debuggerd 收到信号后，就可以生成 tombstone 了。

(5). ptrace detach

完成工作后，需要 detach ptrace，然后发送 sig 18 让其继续奔跑。如果是访问空指针等错误，程序会再次发生异常，由于在信号处理函数里已经将对应信号恢复默认，因此可能会产生 coredump。

详细代码分析如下（省略无关紧要部分）：

```

int status = read_request(fd, &request);
if (!status) {
    XLOG("BOOM: pid=%d uid=%d gid=%d tid=%d\n",
        request.pid, request.uid, request.gid, request.tid);
    /* attach上后会发送sig 19给原进程 */
    ptrace(PTRACE_ATTACH, request.tid, 0, 0);
    TEMP_FAILURE_RETRY(write(fd, "\n", 1));
    if (request.action == DEBUGGER)
        close(fd); fd = -1;
    for (;;) {
        int signal = wait_for_signal(request.tid, &total_sleep_time_usec);
        switch (signal) {
            case SIGSTOP:
                if (request.action == DEBUGGER_ACTION_DUMP_TOMBSTONE) {
                    tombstone_path = engrave_tombstone(request.pid, request.tid);
                } else if (request.action == DEBUGGER_ACTION_DUMP_BACKTRACE) {
                    dump_backtrace(fd, -1, request.pid, request.tid, &detach_f);
                } else {
                    status = ptrace(PTRACE_CONT, request.tid, 0, 0);
                    continue; /* loop again */
                }
                break;
            case SIGILL: case SIGABRT: case SIGSEGV: case SIGPIPE: case SIGSTKFLT:
                kill(request.pid, SIGSTOP); /* 使进程其他线程停止 */
                tombstone_path = engrave_tombstone(request.pid, request.tid, s);
                break;
            default: break;
        }
        break;
    }
    if (request.action == DEBUGGER_ACTION_DUMP_TOMBSTONE) {
        if (tombstone_path)
            write(fd, tombstone_path, strlen(tombstone_path));
        close(fd); fd = -1;
    }
    free(tombstone_path);
    ptrace(PTRACE_DETACH, request.tid, 0, 0);
    kill(request.pid, SIGCONT);
}

```

从socket读取信息

Ptrace attach

等待目标进程的信号，第1次等到的是STOP,第2次才是真正的信号

STOP后会continue，将目标进程上下文切换为异常时刻的

切换OK，直接抓取tombstone

完成后，detach，并发送sig 18使程序继续跑，等待抓取coredump

3. tombstone

(1). 创建 1 个 tombstone 文件。

最多 10 个，如果已存在 10 个，则覆盖最旧的文件。

(2). 版本信息

主要是 fingerprint，可以看出异常版本是 eng 还是 user。

(3). 寄存器信息

主要查看是哪个进程崩溃，信号是什么。寄存器信息需要配合下面的调用栈信息及数据信息结合 GNU 的工具 (objdump -S 反汇编) 分析。

(4). 调用栈信息

这个是最直接可以看出异常的信息。

(5). 其他线程信息

如果异常线程和其他线程有逻辑关系的话，可以查看对应线程的信息。

(6). main log 信息

只会印部分 log 信息，全面的 log 建议还是查看 main log。

详细代码分析如下（省略无关紧要部分）：


```

char* engrave_tombstone(pid_t pid)
{
    .....
    char* path = find_and_open_tombstone(&fd);
    log.tfd = fd;
    log.amfd = activity_manager_connect();
    log.quiet = quiet;
    *detach_failed = dump_crash(&log, pid, tid);
    close(log.amfd);
    close(fd);
    return path;
}

static bool dump_crash(log_t* log, pid_t pid, pid_t tid)
{
    .....
    LOG(log, 1, "**** ***");
    dump_build_info(log);
    dump_revision_info(log);
    dump_thread_info(log, pid, tid, true);
    if (signal)
        dump_fault_addr(log, tid, signal);
    dump_abort_message(log, tid, abort_msg_address);

    ptrace_context_t* context = load_ptrace_context(tid);
    dump_thread(context, log, tid, true, total_sleep_time);

    if (want_logs)
        dump_logs(log, pid, true);

    bool detach_failed = false;
    if (dump_sibling_threads)
        detach_failed = dump_sibling_thread_report(context);

    free_ptrace_context(context);

    if (want_logs)
        dump_logs(log, pid, false);
}

```

找到并打开 tombstone

主要函数

输出fingerprint

输出寄存器等信息

输出调用栈等信息

输出目标进程相关main log

输出其他线程信息

4. 例子解读

至此 tombstone 就完全输出到文件或 main log 里了，大致的格式如下（例子）：

```

Build fingerprint: 'alps/mt6592lte_phone/mt6592lte_phone:4.4.2/fff/1399494:
pid: 704, tid: 709, name: rild >>> /system/bin/rild <<<
signal 6 (SIGABRT), code -6 (SI_TKILL), fault addr -----
Abort message: 'FORTIFY_SOURCE: vsprintf buffer overflow. Calling abort().
r0 00000000 r1 000002c5 r2 00000006 r3 00000000
r4 00000006 r5 00000000 r6 000002c5 r7 0000010c
r8 b6cbf950 r9 b6f4bbd8 sl b6c9de28 fp b6f402fc
ip b6874b9c sp b6874ad0 lr b6efa2e1 pc b6f09224 cpsr 000b0010

backtrace:
#00 pc 00026224 /system/lib/libc.so (tgkill+12)
#01 pc 000172dd /system/lib/libc.so (pthread_kill+48)
#02 pc 000174f1 /system/lib/libc.so (raise+10)
#03 pc 00016225 /system/lib/libc.so
#04 pc 00025ad4 /system/lib/libc.so (abort+4)

```

可以看到 NE 的原因是 sig 6，有人调用了 abort（类似于 assert），原因也印出来了，在 abort message 里。这就需要从调用栈入手分析（当然这个例子看不到是谁调用了 abort()）。

五：流程-调用栈

概要

上一章节讲到 debuggerd 打印调用栈，调用栈是分析问题的关键，因此这边单独拿出来讲解。

目前的 C/C++ 语言的过程调用都需要栈，正在执行的函数有属于自己的栈帧，函数内部的局部变量就放在栈帧里，当然

还会存放函数的返回地址，这样函数执行结束之后才知道返回到哪里。

不同的栈帧关联在一起就会形成一个调用链，最顶端表示当前正在执行的函数，第 2 行表示调用它的函数，以此类推。

下面将讲解 Andriod 上调用栈的形成和解析。

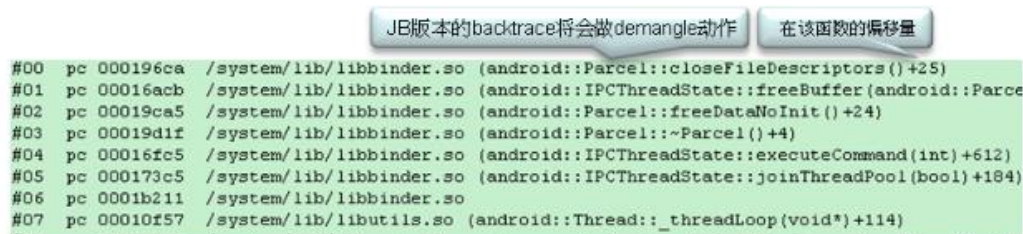
1. 解读

不同版本的 Android 的 debuggerd 打印的调用栈有些差别。

ICS 及以前的版本：



JB 及以后的版本（差别在于函数名做了 demangle，更容易看懂）：



还原有时可能出现问題，只有 2 层：



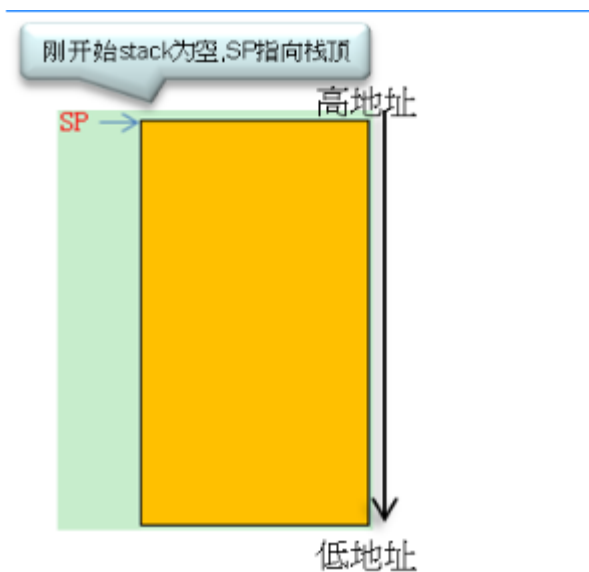
【TIPS】有时不需要拿到含有 debug 信息的库/应用程序而仅通过函数名就可以分析问题了。

2. 栈帧

调用栈的原因只有了解了，才能更好的分析问题。这边会讲解随着代码的执行，栈帧的构造过程。下面以 test 程序为例：

(-1). 程序执行前

内核会分配默认的栈给进程的第 1 个线程使用，此时栈是空的（ARM 的栈是满递减类型），SP 在栈顶：



(0). 执行 main() 函数

根据 main() 函数的汇编代码, 前面做了压栈和保存一些必要的寄存器的动作, 之后 SP 向下 (低地址, 栈是从上往下增长) 移动, 栈布局如下:



对应的汇编指令如下:

```

00008464 0507  main:push  (r0-r2,r14)
00008466 2304      movs  r3,#0x4
00008468 9301      str   r3,[r13,#0x4]
0000846A FFF3F7FF  bl    0x8454 ; a
0000846E FFE9F7FF  bl    0x8444 ; b
00008472 2000      movs  r0,#0x0; return;
00008474 BD0E      pop   (r1-r3,pc)
  
```

函数调用一般用BL指令, 该指令自动将下一条指令的地址保存在R14(LR)中, 然后跳转到相应函数执行, 函数执行时一般都会将LR压栈保存以便函数正确返回

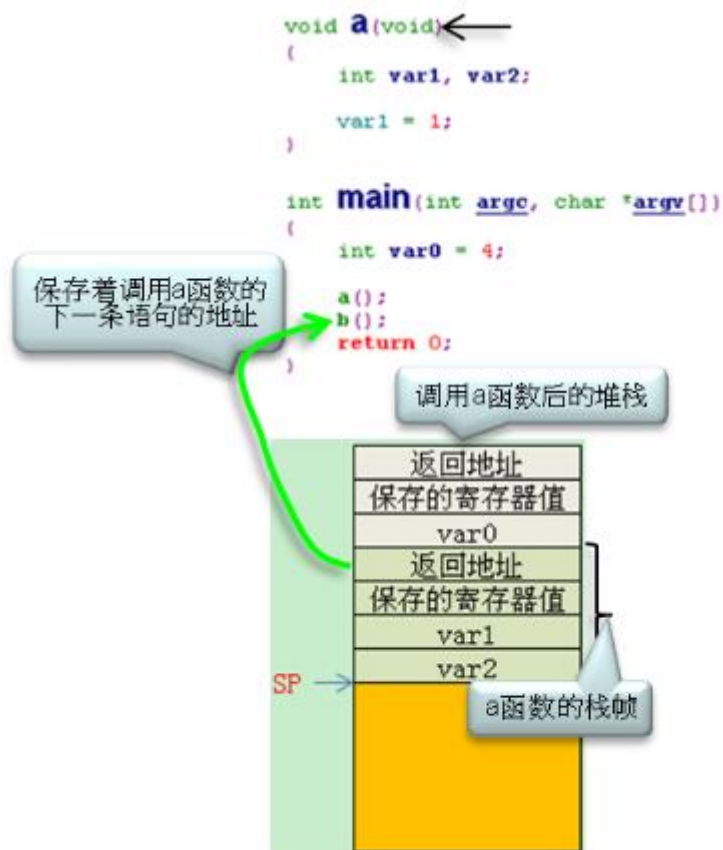
退栈, 将返回地址赋值到PC寄存器

【问题】谁调用了 main() 函数, 是 OS 吗?

【解答】linker 跑完后, 会调用程序的 _start() 而非 main() 函数, 原因是链接脚本定义了入口函数是 _start(), 而且要在 main() 函数前做很多准备工作, 大家可以去看 bionic/libc/arch-arm/bionic/crtbegin.c。

(1). 执行 a() 函数

此时会在 main() 函数之下再压入 a() 函数的栈帧:



对于一个栈帧来说，布局不一定都是一样的，比如有的可能没有保留返回地址，有的没有保存寄存器，编译器会根据情况做到最优化。

比如以下函数就没有保留返回地址，那返回地址记录在哪里呢？ARM 有很多寄存器，有时用不了这么多，因此就保留在寄存器上了，可以减少对内存的访问，提升性能：

编译器优化·没有push LR

00008454	2001	a:	movs	r0, #0x1
00008456	0002		sub	sp, sp, #0x8
00008458	1E43		subs	r3, r0, #0x1
0000845A	9001		str	r0, [r13, #0x4]
0000845C	9300		str	r3, [r13]
0000845E	0002		add	sp, sp, #0x8
00008460	4770		bx	r14

(2). 退出 a() 函数

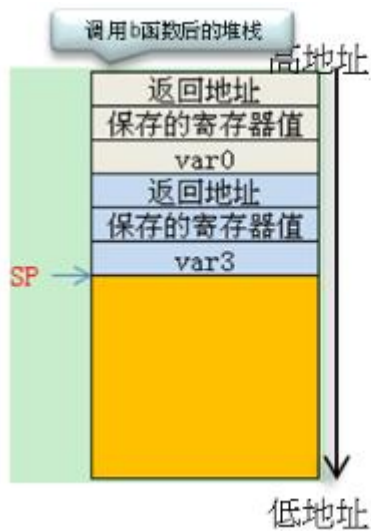
这时 a() 函数的栈帧就要被弹出了，SP 将往上移动。至于之前属于 a() 的栈帧一般还保留着，只不过调用下一个函数后，将被覆盖掉。



【TIPS】局部变量的生命周期就是进入函数到离开函数这段时间。

(3). 调用 b() 函数

原先 a() 函数的栈帧就被覆盖了（从栈里看不出之前执行过的函数的痕迹），SP 向下移动。



(4). 调用 c() 函数

在 b() 函数之下有压入 c() 函数的栈帧，此时有 3 个栈帧了。



(5). 退出c()函数

这里和退出a()函数差不多，这边就省略了。

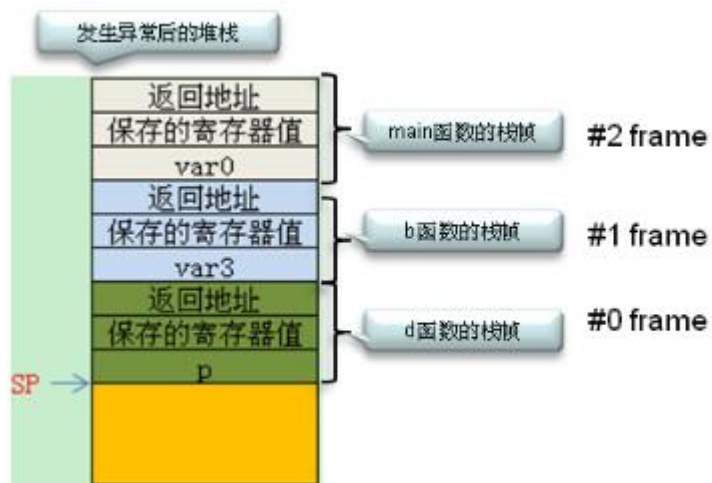
(6). 调用d()函数

这里和调用b()函数差不多。



3. NE 现场

在d()函数里，由于对空指针赋值会引发SIGSEGV，所以最后栈的样子如下：



从tombstone输出的调用栈的栈帧里的地址有特别的含义：


```

void d(void)
{
    char *p = NULL;
    *p = 0x323;
}

void C(void)
{
    int var4 = 6;
}

void b(void)
{
    int var3 = 3;
    c();
    d();
}

int main(int argc, char *argv[])
{
    int var0 = 4;
    a();
    b();
    return 0;
}

```

#0里的PC值比较特殊，表示发生异常的PC值在库里的偏移

#00 pc 00008434 /system/bin/test (d)
 #01 pc 0000844e /system/bin/test (b)
 #02 pc 0000846e /system/bin/test (main)

除了#0，其他的表示调用下一个函数的PC值在库里的偏移

【TIPS】库里的偏移：库默认都加载在 0 地址的，由 OS 随机加载在 mmap 区域，因此实际上库函数的地址都是基址+偏移量，其中的基址就是库加载的地址，这个地址可以从 /proc/\$pid/maps 看到每个库的基址。

这些偏移地址该如何使用呢？

其实我们更想知道该函数所在的源文件和所在行数，而透过 PC 值就可以给出这样的结果，那是谁将 PC 值和源文件/行数关联起来呢？

实际上 Android 已经做好这一切了，编译每一个库和程序，都会为其生成一个带有调试信息的库或程序。这个关联就存放在含有调试信息的库/程序中，可以通过 addr2line 这个工具将 PC 值转化为源文件和行数。

此目录存放的是含有debug信息的应用程序和库

out/target/product/\$proj/symbols
 Name ^
 data
 external
 sbin
 system
 charger
 init

【注意】含有调试信息的库或程序必须和手机里的库或程序是同一编译生成的，否则得到的 PC 值和库里的调试信息不匹配。

【问题】如何确定库或程序含有调试信息呢？

【解答】用 file 命令即可查看：file xxx

如果没有含 debug 信息的话，会显示：

xxx: ELF 32-bit LSB shared object, ARM, version 1 (SYSV), dynamically linked, stripped

含有调试信息的话，则显示：

xxx: ELF 32-bit LSB shared object, ARM, version 1 (SYSV), dynamically linked, not stripped

【TIPS】addr2line 的用法请查看 GNU tools 章节。

概要

当 debuggerd 处理完后，程序再次回到发生异常的环境，此时还会发生 1 次异常，同样的会再次发送信号。

由于 linker 里的 debuggerd_init() 里注册的几个信号，默认行为都会产生 coredump，因此接下来会介绍 coredump 产生以及 db 打包过程。

【TIPS】到这里，目标进程已经收到 3 次同样的信号了!!!

1. 产生 coredump

当一个程序崩溃时，OS 会将该进程的的地址空间保存起来，然后通过工具(GDB, trace32) 离线调试。

有的问题仅仅通过 backtrace 是无法直接定位问题的（指针错误，访问无效内存，内存被踩坏，函数参数错误）。

coredump 默认是关闭的，并且有些参数可以设置它，如下：

(1). 参数

/proc/sys/kernel/core_pattern

这个参数用于设置 coredump 文件的名称，支持的参数有

%p: 添加 pid

%u: 添

加当前 uid

%g: 添加当前 gid

%s: 添加导致产生

core 的信号

%t: 添加 core 文件生成时的 unix 时间

%h: 添加主机名

%e: 添加命令名

在 aed 起来时会对其做初始化

eng build: | /system/bin/aee_core_forwarder /data/core/ %p %s UID=%u GID=%g

user build: /bad_core_pattern

ulimit -a

查看/设置 coredump 文件的大小，默认为 0，也就是不抓 coredump

```
root@android:/ # ulimit -a
ulimit -a
time(cpu-seconds)      unlimited
file(blocks)           unlimited
coredump(blocks)       0
data(KB)               unlimited
```

可以用 ulimit -c <filesize>(KB) 改变大小

【TIPS】当 core_pattern 里有管道时忽略此参数(也就是第 1 个字符为'|')!

/proc/\$pid/coredump_filter

coredump 是抓取进程空间内的内存并保存到文件上，并不是所有内存都需要保存的，你可以通过该参数过滤，只抓取部分内存。

该参数是一个值，每个 bit 位都有对应的含义，用来表示是否抓取这部分内存。

bit0: 私有匿名

bit1: 共享匿名

bit2: 有底层文件的私有映射

bit3: 有底层文件共享映射

bit4: ELF 头

bit5: 私有大尺寸页

bit6: 共享大尺寸页

当前默认值是 0x23，也就是只会抓取：私有匿名/共享匿名/私有大尺寸页

(2). 抓取

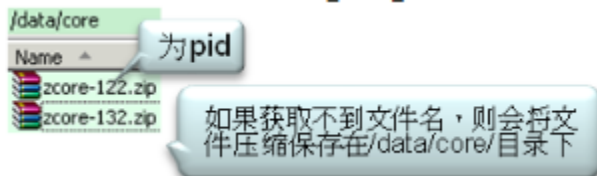
在 eng 版本，/proc/sys/kernel/core_pattern 被设置为 | /system/bin/aee_core_forwarder /data/core/ %p %s UID=%u GID=%g，也就是说进程内存数据会导向 aee_core_forwarder 这个程序。

aee_core_forwarder 会向 aee 询问要保存到哪里，aee 会提供 db 所在路径，然后倒入该路径，最后由 aee 统一压缩为 db。



【TIPS】当 coredump 没有正常生成时可以通过 log 分析问题点

如果 aee 没有反馈, 则 aee_core_forwarder 会保存到/data/core/目录下以 zcore-xxx.zip 文件 (可以用 GAT 解压) 保存。当然了 data 空间很宝贵, 如果生成 coredump 后如果空间小于 4M, 则删除 coredump。



(3). 无 coredump 原因和解决方法

有时候发现进程崩溃了, 但是没有 db 或有 db 但是没有 coredump。这边总结如下:

1. 进程重新注册了几个 NE 的信号, 导致异常时被捕获了, 没有正常的 NE 流程。
2. init 之中发生 NE。init 进程如果有人尝试杀死则会演变为 KE。
3. 存储空间满。
4. 进程异常后其他线程提前退出。
5. fd leak, 导致无法打开 socket, 只能直接触发 coredump 放在/data/core 目录下。
6. user 版本没有开 mtklogger 也是没有 coredump 的。

.....

解决方法

adb shell aee -d coreon 之后删除/system/bin/debuggerd。然后重启机器设置 adb shell echo "/system/bin/aee_core_forwarder /data/core/ %p %s UID=%u GID=%g" > /proc/sys/kernel/core_pattern (注意重启后失效)

复现问题, coredump 可以在 data/core 目录下找到。

(4). 主动触发 coredump

有时为了分析问题, 需要主动触发 coredump, 然后通过工具分析。

你可以通过 adb shell kill -5 \$pid 或连发 3 次 adb shell kill -11 \$pid。

coredump 会在/data/core 或 db 里面。

2. 产生 db

debuggerd 完成之后会通知 aee, aee 就开始了打包 db 的工作。

1 个完整的 NE 的 db, 里面除了 coredump 还有其他文件, 这些文件绝大部分是通过 aee_dumpstate 保存起来的。

```

_exp_main.txt      : exception main log file
_exp_detail.txt    : exception detail dump file
DUMPSYS_ACTIVITY   : dumpsys activity a)
DUMPSYS_GFXINFO    : dumpsys gfxinfo acceleration info (dumpsys gfxinfo)
DUMPSYS_WINDOW     : framework info (dumpsys window)
NE_JBT_TRACES      : Java Backtrace
PROCESS_CMDLINE    : process command line (/proc/pid/cmdline)
PROCESS_COREDUMP    : native program core dump
PROCESS_ENVIRONMENT : process environment (/proc/pid/envIRON)
PROCESS_FILE_STATE  : process file state (/proc/pid/fd)
PROCESS_MAPS        : process memory maps (/proc/pid/maps)
PROCESS_OOM_ADJ     : process oom_adj value (/proc/pid/oom_adj)
PROCESS_OOM_SCORE   : process oom_score value (/proc/pid/oom_score)
PROCESS_SCHED       : process sched info (/proc/pid/sched)
PROCESS_SHOWMAP     : show memory map (showmap)
PROCESS_STATE       : process state (/proc/pid/state)
SCREEN.png          : screen capture bitmap (screencap -p)
SYS_ANDROID_EVENT_LOG : android event log (logcat -b events -v time -d *:v)
SYS_ANDROID_LOG     : android buffer log (logcat -d -v time *:v)
SYS_ANDROID_RADIO_LOG : android buffer log (logcat -b radio -v time -d *:v)
SYS_BACKLIGHTS      : backlight information
SYS_BINDER_INFO     : binder info (/proc/binderinfo)
SYS_BUDDY_INFO      : buddyinfo (/proc/buddyinfo)
SYS_CPU_INFO         : cpu information (top)
SYS_EVENT_LOG_TAGS   : event log tags (/etc/event-log-tags)
SYS_FILE_SYSTEMS     : file system (df)
SYS_FTRACE           : ftrace sched switch tracer

```

异常类型,backtrace等重要信息

异常时的main/radio/event log

整个压缩过程会有 log 印出来, 因此如果有什么问题也可以通过 log 分析。

```

AED : aed_report_archive: pack files in /sdcard/mtklog/aee_exp/temp/db.a26543
AED : put_in_archive: Archive file /sdcard/mtklog/aee_exp/temp/db.a26543/_exp_main.txt, size 4965
AED : put_in_archive: Archive file /sdcard/mtklog/aee_exp/temp/db.a26543/PROCESS_FILE_STATE, size 25043
AED : put_in_archive: Archive file /sdcard/mtklog/aee_exp/temp/db.a26543/_exp_detail.txt, size 473406
AED : put_in_archive: Archive file /sdcard/mtklog/aee_exp/temp/db.a26543/Z2_INTERNAL, size 98
AED : aed_report_archive: pack /sdcard/mtklog/aee_exp/temp/db.a26543/CURRENT.dbg completed
AED : *** INDEX: '/sdcard/mtklog/aee_exp/db.04' from /sdcard/mtklog/aee_exp/temp/db.a26543 ***
AED : aed_report_complete: OK find db slot /sdcard/mtklog/aee_exp/db.04
AED : notify_ok: an broadcast calling for /sdcard/mtklog/aee_exp/db.04
AED : all finished dumping /sdcard/mtklog/aee_exp/db.04.

```

db 中有些文件对分析 NE 是至关重要的, 必须掌握。比如 PROCESS_MAPS, 这只文件就是 /proc/\$pid/maps, 里面是对进程空间的描述:

进程里地址范围	读写权限	空间所映射的文件的主设备号	空间所映射的文件的文件号	空间所映射的文件的路径
00008000-000e0000	r-xp	00000000	b3:05 262	/system/bin/mtkbt
000e0000-000f5000	rw-p	000e5000	b3:05 262	/system/bin/mtkbt
000f5000-001b7000	rw-p	00000000	00:00 0	
01132000-01137000	rw-p	00000000	00:00 0	[heap]
400b8000-400c5000	r--s	00000000	00:0b 1304	/dev/properties (deleted)
400ca000-40128000	r-xp	00000000	b3:05 698	/system/lib/libc.so
40128000-4012b000	rw-p	0005e000	b3:05 698	/system/lib/libc.so
4012b000-40136000	rw-p	00000000	00:00 0	
40136000-4014b000	r-xp	00000000	b3:05 803	/system/lib/libm.so
4014b000-4014c000	rw-p	00015000	b3:05 803	/system/lib/libm.so
40161000-40162000	r-xp	00000000	b3:05 889	/system/lib/libstdc++.so
40162000-40163000	rw-p	00001000	b3:05 889	/system/lib/libstdc++.so
b0001000-b000a000	r-xp	00001000	b3:05 234	/system/bin/linker
b000a000-b000b000	rw-p	0000a000	b3:05 234	/system/bin/linker
b000b000-b001c000	rw-p	00000000	00:00 0	
be936000-be957000	rw-p	00000000	00:00 0	[stack]
ffff0000-ffff1000	r-xp	00000000	00:00 0	[vectors]

虚拟内存的权限:
r=读
w=写
x=可执行
s=共享
p=私有

空间所映射的文件的主设备号

空间所映射的文件的文件号

空间所映射的文件的路径

每一项都与 vm_area_struct 结构成员对应

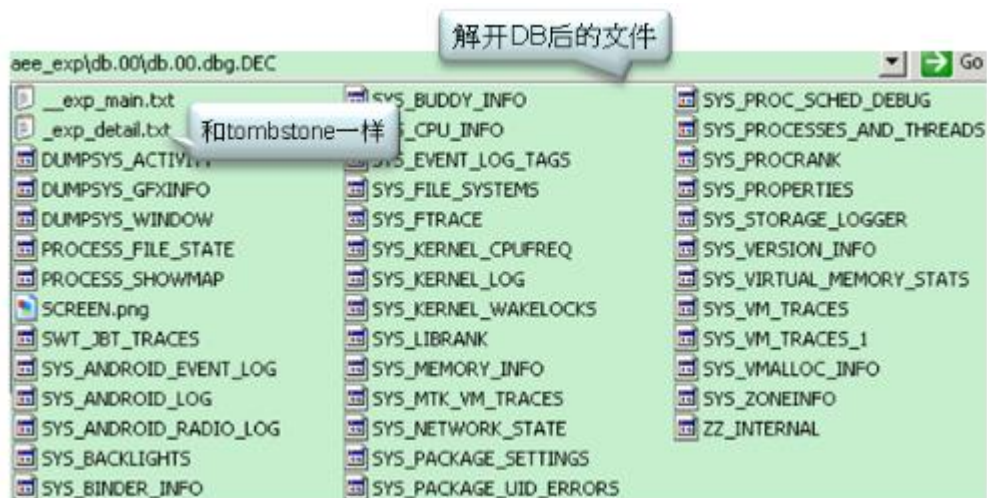
0 表示没有节点映射到内存

Virt size: 映射到 user space 的内存总和

1). db 解包

db 需要 GAT 工具才可以解包，更多细节请查看 DCC 上的 GAT 使用文档。

大致结果如下：



(2). db 相关配置

db 个数：user 版本：4 个，eng 版本：20 个

coredump 只有在以下设置才会抓取：

eng 版本

user 版本+打开 mtklogger

详情请看 DCC 上的文档：MediaTek Logging SOP.pptx

结语

至此，整个 NE 的流程就结束了，那么大家就应该可以简单分析 NE 产生的问题了。coredump 分析将会在下一章节讲解。

七：流程-debuggerd on android P

1. tombstoned 服务

P 版本上移除掉了 debuggerd 进程，多了一个 tombstoned 进程

在系统启动过程中，会将 tombstoned trigger 起来。tombstoned 会创建 socket，然后监听。

等待别人通过 socket 来请求服务。tombstoned 返回 backtrace 文件或者 tombstoned 文件的文件描述符。

具体 code 是在 system/core/debuggerd/tombstoned/tombstoned.cpp

```
int main(int, char* []) {
398     umask(0137);
399
400     // Don't try to connect to ourselves if we crash.
401     struct sigaction action = {};
402     action.sa_handler = [](int signal) {
403         LOG(ERROR) << "received fatal signal " << signal;
404         _exit(1);
405     };
406     debuggerd_register_handlers(&action);
407 }
```



```

408     int intercept_socket = android_get_control_socket(kTombstonedInterceptSocketName);
409     int crash_socket = android_get_control_socket(kTombstonedCrashSocketName);
410
411     if (intercept_socket == -1 || crash_socket == -1) {
412         PLOG(FATAL) << "failed to get socket from init";
413     }
414
415     evutil_make_socket_nonblocking(intercept_socket);
416     evutil_make_socket_nonblocking(crash_socket);
417
418     event_base* base = event_base_new();
419     if (!base) {
420         LOG(FATAL) << "failed to create event_base";
421     }
422
423     intercept_manager = new InterceptManager(base, intercept_socket);
424
425     evconnlistener* tombstone_listener =
426         evconnlistener_new(base, crash_accept_cb, CrashQueue::for_tombstones(), LEV_OPT_CLOSE_ON_FREE,
427             -1 /* backlog */, crash_socket);
428     if (!tombstone_listener) {
429         LOG(FATAL) << "failed to create evconnlistener for tombstones.";
430     }
431
432     if (kJavaTraceDumpsEnabled) {
433         const int java_trace_socket = android_get_control_socket(kTombstonedJavaTraceSocketName);
434         if (java_trace_socket == -1) {
435             PLOG(FATAL) << "failed to get socket from init";
436         }
437
438         evutil_make_socket_nonblocking(java_trace_socket);
439         evconnlistener* java_trace_listener =
440             evconnlistener_new(base, crash_accept_cb, CrashQueue::for_anrs(), LEV_OPT_CLOSE_ON_FREE,
441                 -1 /* backlog */, java_trace_socket);
442         if (!java_trace_listener) {
443             LOG(FATAL) << "failed to create evconnlistener for java traces.";
444         }
445     }
446
447     LOG(INFO) << "tombstoned successfully initialized";
448     event_base_dispatch(base);

```

2. debuggerd_signal_handler 处理流程

进程启动过程中，依然是在 linker 来 call 到 debuggerd_init(), 注册 signal handle。这和之前版本一样。

当进程收到 signal, call 到 signal 处理函数, debugger_signal_handler 处理流程大致如下:

进阶篇: coredump 分析

1. GNU tools

2. 概要

Android 编译工具使用了 gcc, 因此了解和使用 gnu toolchain 可以在 debug 时更加方便。

工具目录(这里只列了 ARM 平台相关的工具)

ARM32 位版本: prebuilts/linux-x86/gcc/arm/arm-linux-androideabi-\$version/bin

3. ARM64 位版本: prebuilts/linux-x86/gcc/aarch64/aarch64-linux-android-\$version/bin

为了方便使用, 可以将目录添加到环境变量中

编辑 \$HOME/.bash_profile 文件, 在最后添加以下语句即可将 \$HOME/bin 加入环境变量, \$HOME/bin 也可以改为你想添加的路径:

```
PATH=$PATH:$HOME/bin
export PATH
```

1. addr2line

将地址转换为地址所在的文件及行数(显示所在函数)

使用方法: arm-linux-androideabi-addr2line [option(s)] [addr(s)]

(1). 参数 (常用)

-e --exe=<executable>: 设置要查询地址的文件(默认: a.out)

一般是 *.so/*.a 和可执行程序

此文件必须带有 debug 信息, 在 android codebase 里是放在 out/target/product/\$project/symbols 目

录下

-f - functions: 显示地址所在的函数名

-C --demangle[=style]: 反重整函数名为可读方式

自动识别格式, C++ 函数才需要此参数

(2). 例子

```
arm-linux-androideabi-addr2line -e libc.so -f -C 0x23234
```

```
wscoll
```

```
alps/bionic/libc/wchar/wscoll.c:37
```

(3). 什么情况下需要用到?

发生 NE 后, 会生成 tombstones/tombstones_xx 文件或 aee_exp 里的 db 解开之后的 __exp_main.txt, 里面有 backtrace 信息, 就可以通过 addr2line 分析出哪个文件哪行哪个函数(注意用 -e 载入的文件必须和手机的 bin 档同一次编译生成, 否则地址和符号可能不一致)

2. nm

```
arm-linux-androideabi-nm [option(s)] [file(s)]
```

列出该文件的符号(函数, 变量, 文件等), 包含名字、地址、大小

(1). 参数 (常用)

-C, --demangle[=STYLE]: 反重整符号为可读方式

自动识别格式

-e --exe=<executable>: 设置要查询地址的文件(默认: a.out)

一般是 *.so 和可执行程序

-D, --dynamic: 只显示动态符号

-g, --extern-only: 只显示外部符号
-l, --line-numbers: 多显示符号所在文件和行数
-S, --print-size: 多显示符号的大小
-u, --undefined-only: 只显示未定义的符号

(2). 符号类型 (常用)

小写表示是本地符号, 大写表示全局符号 (external)

A: 符号值是绝对的。在进一步的连接中, 不会被改变 (absolute)

B: 符号位于未初始化数据段 (BSS section)

C: 共用 (common) 符号。共用符号是未初始化的数据。在连接时, 多个共用符号可能采用一个同样的名字, 如果这个符号在某个地方被定义, 共用符号被认为是未定义的引用

D: 已初始化数据段的符号 (data section)

F: 源文件名称符号

R: 只读数据段符号. (定义为 const 的变量)

T: 代码段的符号 (text section)

U: 未定义符号

?: 未知符号类型, 或者目标文件特有的符号类型

(3). 例子

```
arm-linux-androideabi-nm -g test
00009018 D __CTOR_LIST__
00009010 T __FINI_ARRAY__
00009008 T __INIT_ARRAY__
U __aeabi_unwind_app_pr0
.....
```

3. objdump

arm-linux-androideabi-objdump <option(s)> <file(s)>

查看对象文件 (*.so/*.a 或应用程序) 的内容信息

(1). 参数 (常用)

至少需要一个以下的参数

-a, --archive-headers: 显示库 (*.a) 成员信息

-f, --file-headers: 显示 obj 中每个文件的整体头部摘要信息

-h, --[section]-headers: 显示目标文件各个 section 的头部摘要信息

-x, --all-headers: 显示所有头部摘要信息

-d, --disassemble: 反汇编代码段

-D, --disassemble-all: 反汇编所有段

-S, --source: 反汇编出源代码, 额外有 debug 信息, 隐含 -d, 如果编译时有 -g, 效果更明显

-t, --syms: 显示符号表

-r, --reloc: 显示重定位记录

-C, --demangle[=STYLE]: 反重整符号为可读方式

自动识别格式

(2). 例子

```
arm-linux-androideabi-objdump -S libstdc++.so > disas.txt
```

(3). 什么情况下需要用到?

当发生 NE 后, 拿到 backtrace 可以查看对应地址的汇编代码。

4. readelf

```
arm-linux-androideabi-readelf <option(s)> elf-file(s)
```

查看 elf 文件(*.so/*.a 或应用程序)的内容信息

(1). 参数 (常用)

- a, --all: 显示所有可显示的内容
- h --file-header: 显示 ELF 文件头
- l --segments: 显示程序头组
- S --sections: 显示节头组
- t: 显示节头细节
- e --headers: 等效于-h -l -S
- s --syms: 显示符号表
- n --notes: 显示内核说明
- r --relocs: 显示重定位信息
- u --unwind: 显示解栈信息
- d --dynamic: 显示动态节
- p - string-dump=<num|name>: 以字符串的方式显示节
- W --wide: 允许一行显示超过 80 个字符

(2). 例子

```
arm-linux-androideabi-readelf -a -W adb > 1.txt
```

(3). 什么情况下需要用到?

学习/查看 ELF 结构。

5. c++filt

```
arm-linux-androideabi-c++filt <function name>
```

反重整 C++符号为可读方式

(1). 例子

```
arm-linux-androideabi-c++filt
```

```
_ZN20android_audio_legacy22AudioPolicyManagerBase17setSystemPropertyEPKcS2
```

```
android_audio_legacy::AudioPolicyManagerBase::setSystemProperty(char const*, char const*)
```

(2). 什么情况下需要用到?

通过 GAT 解开 db 后(输入 symbols 目录), 如果是 NE 的话, 可以查看解开的 C++的对应的 symbols 是否和 c++filt 的一样, 不一样则 symbols 目录和手机 bin 档不匹配。

2. AAPCS 标准

概要

使用工具分析 core dump, 不可避免要接触 ARM 汇编。汇编里的指令的含义可以查看 spec 了解, 而函数参数的传递方法, 栈的布局, 寄存器的规划是查看汇编的重要部分, 这部分由 ARM AAPCS 定义。熟悉它才能理解和分析 core dump。

1. Procedure Call Standard for the ARM Architecture

The layout of data

Layout of the stack and calling between functions with public interfaces

Variations available for processor extensions, or when the execution environment restricts the addressing model

The C and C++ language bindings for plain data types

熟悉 AAPCS 后, 查看汇编代码将有很大帮助。

2. 文档来源

ARM 官网 <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faq5/index.html>

ARM 软件开发工具 => ABI for the ARM Architecture => Procedure Call Standard for the ARM Architecture

3. Core registers

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

R0-R3 (a1-a4)

子程序间通过 r0-r3 来传递子程序前 4 个参数(剩余参数通过压栈传递, 8 个字节的 double 类型由两个寄存器一起传递)

子程序返回结果, 也是使用 r0-r3 来传递, 如果组合类型(编译时不确定长度的)超过 4 字节的将结果存放在 memory, 然后返回地址

R4-R8, R10 and R11 (v1-v5, v7 and v8)

用于保存局部变量(thumb 用 R4-R7 但不局限于 R4-R7)

R9

平台相关, 比如:

在数据位置无关模型里作为: static base(SB)

作为 TLS 指针: thread register(TR)

或是作为普通的 v6

R12

r12 用作子程序间(子程序如果太远需要胶合代码辅助跳转)scratch 寄存器, 别名为 ip

也可以用来保存中间值

R13-R15

特殊用途, 分别为 SP(栈指针), LR(保存子程序的返回地址), PC(程序计数器)

LR 保存后, 可以客串下保存中间值

在调用子程序前后, 保证 R4-R8, R10, R11 和 SP(如果 R9 作为 v6, 则包括 R9)的值不变

如果在子程序内有使用以上寄存器，那么必须在子程序头做压栈保存寄存器值，在退出之程序前做出栈回复寄存器值。

4. 堆栈/子程序调用

堆栈为FD类型，对堆栈的操作是8字节对齐的。使用 stmb/ldmia 批量内存访问指令来操作FD堆栈

LDRD/STRD 要求数据栈是8字节对齐的，以提高数据的传送速度

比如：STMFD sp!, {R4-R11, LR, PC}，PC是不需要压栈保存的，这里压栈仅仅是为了保证8字节边界对齐

子程序调用

需要完成以下动作

LR[31:1]保存返回地址，LR[0]保存返回状态：0：ARM，1：thumb

PC指向子程序头地址

可以直接用BL指令也可以用多条指令合成

ARM状态下：MOV LR, PC; BX R4;

3. GDB 调试

概要

coredump 是 linux 原生的概念，目前有很多工具可以支持 coredump 调试。其中 gdb/trace32 都可以支持。

coredump 包含进程空间的内存，如果在加上含有调试信息的 lib/程序，那么可以还原出当时异常的场景，这时你可以查看寄存器内容，调用栈，变量和内存等等。这对分析问题非常有帮助。

下面我们会一一介绍 gdb 和 trace32 如何调试 coredump。

1. gdb (GNU debugger)

(1). 概述

GDB 是 GNU 开源组织发布的一个强大的 UNIX 下的程序调试工具

官网：<http://www.gnu.org/software/gdb/>（可以下载到工具和文档）

(2). 功能

启动或连接程序，可以按照你的自定义的要求随心所欲的运行程序。

可让被调试的程序在你所指定的调置的断点处停住。（断点可以是条件表达式）。

当程序被停住时，可以检查此时你的程序中所发生的事。

动态的改变你程序的执行环境。

(3). 调试方法

在线调试（需要在 eng build 版本）

1. 可以用 gdb 直接启动一个程序调试。
2. 对一个已经运行的程序用 gdb attach 调试。

离线调试（借助 coredump）

就是本章节的重点。

(4). 工具来源

Android NDK：<http://developer.android.com/sdk/ndk/index.html>。

在 ndk 安装目录下的

toolchains/arm-linux-androideabi-\$version/prebuilts/linux-x86/bin/arm-linux-androideabi-gdb

另外还可以从 codebase 里找到，在

alps/prebuilts/gcc/linux-x86/arm/arm-linux-androideabi-\$version/bin/arm-linux-androideabi-gdb

【注意】不要使用 PC 预装的 gdb，android 有对 gdb 做修改。

2. 离线调试

gdb 离线调试 coredump，一定要有这个程序的 symbols 文件才行。你可以这样启动 gdb：

arm-linux-androideabi-gdb <program> PROCESS_COREDUMP

gdb 有一些启动参数可以配置：

- symbols <file>或-s <file>: 从指定文件中读取符号表。
- se file: 从指定文件中读取符号表信息，并把他用在可执行文件中。
- core <file>或-c <file>: 调试时 core dump 的 core 文件。
- directory <directory>或-d <directory>: 加入一个源文件的搜索路径。默认搜索路径是环境变量中 PATH 所定义的路径。

当然了，简单使用 gdb 的话，直接 arm-linux-androideabi-gdb 即可启动，然后在里面输入对应的命令。

进入 gdb 之后，会有类似 shell 的窗口，你可以输入任何 gdb 的命令，gdb 的调试就是在这样的窗口上进行的。下一章节我们会介绍 gdb 常用的几个命令。

3. 命令

gdb 的命令可以使用 help 命令来查看，如下所示

命令有很多，gdb 把之分成许多个种类。help 命令只是例出 gdb 的命令种类，如果要看种类中的命令，可以使用 help <class>命令，如：help breakpoints，查看设置断点的所有命令。也可以直接 help <command>来查看命令的帮助。

在 gdb 中输入命令时，可以不用打全命令，只用打命令的前几个字符就可以了，当然命令的前几个字符应该要标志着一个唯一的命令，在 Linux 下，你可以敲击两次 TAB 键来补齐命令的全称，如果有重复的，那么 gdb 会将其例出来。

以下列出常用命令（更多的命令请查看 gdb 官网的用户手册）：

命令	缩写	描述
backtrace	bt	Prints a stack trace
display		Displays the value of an expression every time execution stops
finish		Runs to the end of the function and displays return values of that function
jump		Jumps to an address and continues the execution there
list	l	Lists the next 10 lines
next	n	Steps to the next machine language instruction
print	p	Prints the value of an expression
run	r	Runs the current program from the start
set		Changes the value of a variable

(1). 查看栈信息

当程序被停住了，你需要做的第一件事就是查看程序是在哪里停住的。

backtrace 或 bt <n>

- 无 n: 打印当前的函数调用栈的所有信息。
- n 是一个正整数: 只打印栈顶上 n 层的栈信息。
- n 是一个负整数: 只打印栈底下 |n| 层的栈信息。

frame <n>或 f <n> （n 是一个从 0 开始的整数，是栈中的层编号）

如果你要查看某一层的信息，你需要在切换当前的栈，一般来说，程序停止时，最顶层的栈就是当前栈。

up <n>或 down <n>

向栈的上面/下面移动 n 层，可以不打 n，表示向上/下移动一层。

info frame 或 info f

打印出更为详细的当前栈层的信息（目前的函数是由什么样的程序语言写成的、函数参数地址及值、局部变量的地址等等）。

info args

打印出当前函数的参数名及其值。

info locals

打印出当前函数中所有局部变量及其值。

(2). 查看源程序

显示源代码 `list` 或 `l`, `list` 后面可以跟以下参数:

<linenum> 行号。
<+offset> 当前行号的正偏移量。
<-offset> 当前行号的负偏移量。
<filename:linenum> 哪个文件的哪一行。
<function> 函数名。
<filename:function> 哪个文件中的哪个函数。
<*address> 程序运行时的语句在内存中的地址。

指定源文件的路径:

`directory <dirname ... >`或 `dir <dirname ... >`

加一个源文件路径到当前路径的前面, 如有多个路径, UNIX 下你可以使用“:”, Windows 下你可以使用“;”。

`directory`

清除所有的自定义的源文件搜索路径信息。

`show directories`

显示定义了的源文件搜索路径。

(3). 查看变量

在你调试程序时, 当程序被停住时, 你可以使用 `print(p)` 命令查看变量

`print [<f>] <expr>`

<expr>是表达式, 是你所调试的程序的语言的表达式 (GDB 可以调试多种编程语言)。

可以是当前程序运行中的 `const` 常量、变量、函数等 (不能用宏)。

有几种操作符, 它们可以用在任何一种语言中。

:: 指定一个在文件或是一个函数中的变量, 比如 `'file'::variable`, `function::variable`。

{<type>} <addr> 表示一个指向内存地址<addr>的类型为 `type` 的一个对象。

如果你的程序编译时开启了优化选项, 那么在用 GDB 调试被优化过的程序时, 可能会发生某些变量不能访问, 或是取值错误码的情况。

gcc 可以加 `-gstabs` 解决此问题。

<f>是输出的格式

x 按十六进制格式显示变量。

d 按十进制格式显示变量。

u 按十六进制格式显示无符号整型。

o 按八进制格式显示变量。

t 按二进制格式显示变量。

c 按字符格式显示变量。

f 按浮点数格式显示变量。

(4). 查看内存

`examine(x)` 命令查看内存: `x/<n/f/u> <addr>`

n 是一个正整数, 表示显示内存的长度, 也就是说从当前地址向后显示几个地址的内容。

f 表示显示的格式。如果地址所指的是字符串, 那么格式可以是 `s`, 如果指令地址, 那么格式可以是 `I`。

u 表示从当前地址往后请求的字节数, 如果不指定的话, 默认是 4 个 bytes。也可以用下面的字符来代替, `b` 表示单字节, `h` 表示双字节, `w` 表示四字节, `g` 表示八字节。

(5). 查看寄存器和线程

`info registers [<regname ...>]`或 `info reg [<regname ...>]`

查看寄存器的情况（除了浮点寄存器）。

<regname ...>查看所指定的寄存器的情况。

同样可以使用 print 命令来访问寄存器的情况，只需要在寄存器名字前加一个\$符号就可以：p \$sp。

```
info all-registers
```

查看所有寄存器的情况（包括浮点寄存器）。

```
info thread
```

查看所有 thread。

```
thread <threadnum>
```

切换到某个 thread。

4. 例子

以某个 app 为例子，发生了 NE，产生了 db。先用 GAT 解开 db，拿到 PROCESS_COREDUMP，放到/home/db/

假设：

gdb 路径：

/home/alps/prebuilts/gcc/linux-x86/arm/arm-linux-androideabi-4.8/bin/arm-linux-androideabi-gdb。

symbols 路径：/home/alps/out/target/product/\$proj/symbols。

开始启动 gdb，app 对应的程序为 system/bin/app_process：

```
cd /home/db/
```

```
/home/alps/prebuilts/gcc/linux-x86/arm/arm-linux-androideabi-4.8/bin/arm-linux-androideabi-gdb
```

```
/home/alps/out/target/product/$proj/symbols/system/bin/app_process PROCESS_COREDUMP
```

这样就进入 gdb 的命令行了，下面需要设置 symbols 搜索路径，设置后 gdb 会自动加载所需的 lib 库：

```
set solib-absolute-prefix /home/alps/out/target/product/$proj/symbols
```

```
set solib-search-path /home/alps/out/target/product/$proj/symbols/system/lib
```

到这里就完成了 gdb 的启动和加载，之后就可以自由使用各种命令分析 NE 了，比如 bt，info registers 等。更多使用 gdb 的命令或技巧请多查看官方文档或网络上有关 gdb 的技术分享。

4. trace32 调试

Trace32 有专门的文章介绍，请参考：

- [MediaTek On-Line](#)> [Quick Start](#)> Trace32 使用教程
- 《Trace32 使用教程》里有 NE 分析章节，里面有讲解 NE cmm 脚本。

扩展篇：编译与加载

1. native 编译

概要

从前面的章节我们了解到，Android native 程序都是用 arm-linux-androideabi-gcc 编译的。

了解 gcc 如何将*.c/cpp 编译成*.o 再将其链接为可执行程序或/lib 库，有助于我们将 native 从编译/加载/执行到崩溃一条路贯通起来。

Android 的 Makefile 只需要将 source file 填入 LOCAL_SRC_FILES，然后 include \$(BUILD_SHARED_LIBRARY)或 \$(BUILD_EXECUTABLE)就可以将*.c/cpp/s 编译为动态库或可执行程序。其中编译系统做了很多工作，我们不会介绍其中的原因，想要了解的话应该看 build 相关的文档。

本章节会讲解 gcc 部分参数的原理及链接的过程。

1. 编译为 obj

在 build/core/definitions.mk 有定义 transform-c-or-s-to-o-no-deps 和 transform-cpp-to-o，分别将每个 *.c/s 和 *.cpp 编译成 *.o，里面传了很多参数给 gcc，其中 -fpic -fPIE 和 -fstack-protector 是下面会讲解的。

(1). -fpic -fPIE

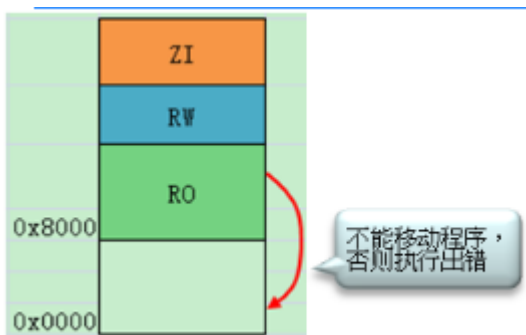
PIC 是 Position-Independent Code 的缩写，经常被用在共享库中，这样就能将相同的库代码为每个程序映射到一个位置，不用担心覆盖掉其他程序或共享库。

PIE 是 Position-Independent-Executable 的缩写，只能应用在可执行程序中。PIE 和 PIC 很像，但做了一些调整（不用 PLT，使用 PC 相关的重定位）。-fPIE 给编译用，-pie 给链接(ld)用。

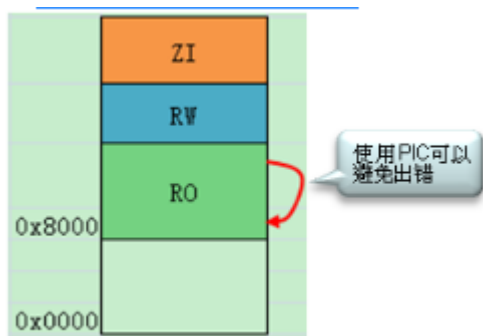
例如，一个程序没有使用 PIC 被链接到 0 地址，那么系统将其加载到 0 地址，程序可以正常运行：



如果系统将其加载到 0x8000，程序会异常：



如果开启 PIC，这会是这样：



使用了什么技术可以达到这样呢？

a. 对于访问自己的函数等可以使用以 PC+偏移量的方式达到效果：

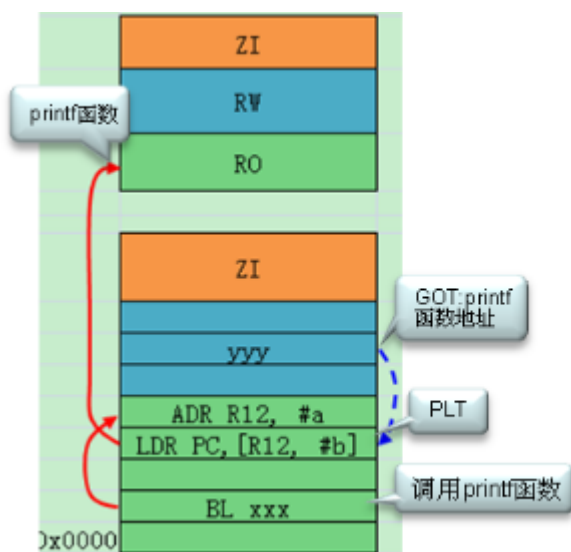


b. 而访问外部共享库等需要 GOT/PLT 支持才行。

PLT (Procedure Linkage Table): 由一个个桩函数组成的跳转表

GOT (Global Offset Table): 由一个个偏移量组成的偏移表, 决定 PLT 跳转的位置 (由 linker 修改)

要调用外部的函数会先跳转到 PLT, PLT 再从 GOT 获取实际的位置, 再跳转过去。PLT/GOT 类似一座桥:



实例分析:

某个程序的函数 `xxx()` 调用外部函数 `__futex_wait()`, `xxx()` 在 .text 段里会直接跳转到 `__futex_wait` 的 PLT 里, PLT 会读取对应 GOT 的地址, 直接跳转过去:

```
.text
0000400: BL 0000814 <= __futex_wait 桩函数

.plt
0000800: E52DE004 str    r14, [r12, #0x4]!
0000804: E59FE004 ldr    r14, 0x810      ; r14 = 0x17C8
0000808: E08FE00E add    r14, pc, r14
000080C: E5BEF008 add    pc, [r14, #0x8]! <= pc = [0x1FE8]
0000810: 000017C8

0000814: E28FC600 adr    r12, 0x81C
0000818: E28CC801 add    r12, r12, #0x1000
000081C: E50CF7C8 ldr    pc, [r12, #0x7C8]! <= pc = [0x1FE4] <= __futex_wait

0000820: E28FC600 adr    r12, 0x828
0000824: E28CC801 add    r12, r12, #0x1000
0000828: E50CF7C8 ldr    pc, [r12, #0x7C8]! <= pc = [0x1FE8] <= __futex_wait

000082C: E28FC600 adr    r12, 0x834
0000830: E28CC801 add    r12, r12, #0x1000
0000834: E50CF7C8 ldr    pc, [r12, #0x7C8]! <= pc = [0x1FEC] <= malloc

.got
PROGBITS 00001fd4 000fd4
00001FD4: 00000000 <= __sf
00001FD8: 00000000 <= _GLOBAL_OFFSET_TABLE_ <= .got
00001FDC: 00000000
00001FE0: 00000000
00001FE4: 00000000 <= .plt <= __futex_wait
00001FE8: 00000000 <= .plt <= __futex_wait
00001FEC: 00000000 <= .plt <= malloc
00001FF0: 00000000 <= .plt <= abort
00001FF4: 00000000 <= .plt <= free
00001FF8: 00000000 <= .plt <= fwrite
00001FFC: 00000000 <= .plt <= __cxa_finalize
```

GOT 表的地址会在 linker 加载库时被修改, 因为所有的库都是由 linker 加载的, 也只有 linker 知道 `__futex_wait()` 被加载到哪里去了。

这里 linker 会有两种做法:

第 1: 在加载库时就把 GOT 全部修改好, 后面程序调用就无需修改, 加快了程序的运行, 但启动较慢。

这个就是所谓立即绑定, Android 目前使用这种方法。

第 2: 先不改 GOT, 此时 GOT 默认会指向第 1 个 PLT, 这个 PLT 会比较特殊, linker 会修改这个 PLT 对

应的 GOT，使它指向 linker 某个函数，那这样只有 GOT 没有被修改下会自动跳转到 linker，由 linker 再去单独修改这个 GOT，达到按需修改，相比第 1 种，可以快速启动，但可能影响运行。这种也称为迟绑定技术（Lazy binding）。

大家看到这里肯定会想，方法有了，那 linker 将 GOT 修改为什么值呢？其实在动态库里有 .rel.dyn 和 .rel.plt 段，里面包含了哪里要修改，修改的方法和对应的符号，linker 就可以轻松完成这件艰巨的任务：

重定位信息

Relocation section '.rel.dyn' at offset 0x7a0 contains 5 entries:				
Offset	Info	Type	Syn. Value	Symbol's Name
000007a0:000009e0	00000017	R_ARM_RELATIVE		
000007a8:00001ed0	00000017	R_ARM_RELATIVE		
000007b0:00001ed4	00000017	R_ARM_RELATIVE		
000007b8:00001ee4	00000017	R_ARM_RELATIVE		
000007c0:00001fd4	00001615	R_ARM_GLOB_DAT	00000000	__SF

Relocation section '.rel.plt' at offset 0x7c8 contains 7 entries:				
Offset	Info	Type	Syn. Value	Symbol's Name
000007c8:00001fe4	00000216	R_ARM_JUMP_SLOT	00000000	__futex_wait
000007d0:00001fe8	00000516	R_ARM_JUMP_SLOT	00000000	__futex_wake
000007d8:00001fec	00000816	R_ARM_JUMP_SLOT	00000000	malloc
000007e0:00001ff0	00000916	R_ARM_JUMP_SLOT	00000000	
000007e8:00001ff4	00000c16	R_ARM_JUMP_SLOT	00000000	
000007f0:00001ff8	00001516	R_ARM_JUMP_SLOT	00000000	
000007f8:00001ffc	00002416	R_ARM_JUMP_SLOT	00000000	__cxa_finalize

Offset指出要修改的位置

Info指出要修改的类型和对应的符号

Linker查找对应符号的信息(地址信息等)

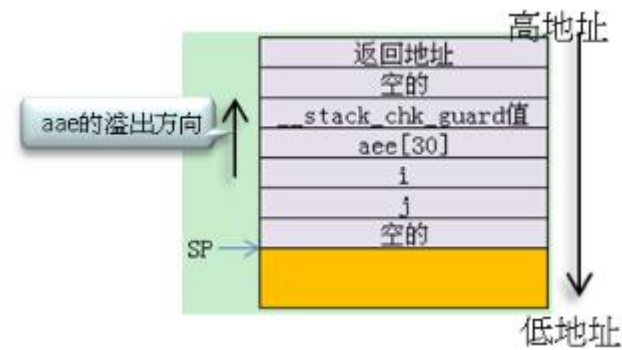
(2). -fstack-protector

顾名思义就是保护堆栈，每一个函数在运行时都有自己的栈帧，如果代码没有写好，很可能将自己甚至是其他的栈帧踩坏，那如何防护呢？简单的方法就是在栈帧头部也就是在局部变量开始之前多存储一个 __stack_chk_guard 值，用于在函数返回前取出来和 __stack_chk_guard 做对比，失败则调用 __stack_chk_fail 函数，这个就是该参数完成的行为。

以下是示意图：

```
void XXX(void)
{
    volatile int i = 0x40, j = 0;
    volatile char aae[0x30];

    aae[1] = 1;
}
```



打开该功能后，编译会自动插入所需代码：



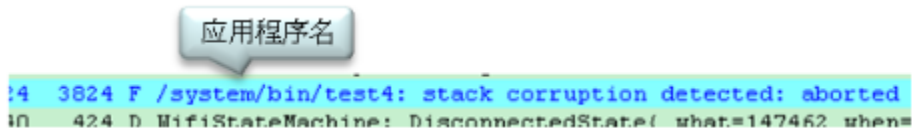
当然该参数不是所有的函数都保护的，一般踩坏栈帧多半是 for/while 循环迭代数组，所以可以如下配置：

-fstack-protector：只保护有定义局部数组且数组个数大于 8 的函数。

--param=ssp-buffer-size=xx：在-fstack-protector 基础上增加该参数可以修改数组个数。

-fstack-protector-all：保护所有函数。

发生 stack corruption 时会跑到__stack_chk_fail()，印出相关信息：



2. 静态链接

build/core/combo/TARGET_linux-arm.mk 里有定义 transform-o-to-static-executable-inner，将*.o 链接成静态可执行程序，静态可执行程序是一个完整的程序，不需要额外的共享库即可执行，比如/init,/sbin/adbd 等。

链接器用的是 arm-linux-androideabi-g++，主要的参数和介绍如下（这里以 test.c 编译为 test 为例子）：

```

-Bstatic  ##表示静态链接##
out/target/product/$proj/obj/lib/crtbegin_static.o
out/target/product/$proj/obj/EXECUTABLES/test_intermediates/test.o
prebuilts/gcc/linux-x86/arm/arm-linux-androideabi-$version/lib/gcc/arm-linux-androideabi/$version-google/armv7-a/libgcc.a
out/target/product/$proj/obj/lib/crtend_android.o

```

大家注意到没有，除了 test.o 居然还链接了 crtbegin_static.o/libgcc.a 和 crtend_android.o，这是怎么回事？？

libgcc.a 后面会详细讲，而 crtbegin_static.o/crtend_android.o 对应代码是（从 bionic/libc/Andoird.mk 看出）bionic/libc/arch-arm/bionic 目录下的 crtbegin.c 和 crtend.s，其中定义了.preinit_array, .init_array 和 fini_array 段以及_start 函数。

再查看 gcc 默认的链接脚本

（prebuilts/gcc/linux-x86/arm/arm-linux-androideabi-\$version/arm-linux-androideabi/lib/ldscripts/armelf_linux_eabi.x）：

```

/* Default linker script, for normal executables */
OUTPUT_FORMAT("elf32-littlearm", "elf32-bigarm",
              "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)
SEARCH_DIR("/usr/local/lib"); SEARCH_DIR("/lib"); SEARCH_DIR("/usr/lib");
SECTIONS
{
    /* Read-only sections, merged into text segment: */
    PROVIDE (__executable_start = SEGMENT_START("text-segment", 0x00008000)); .
    .interp      : { *(.interp) }
    .note.gnu.build-id : { *(.note.gnu.build-id) }
    .hash        : { *(.hash) }
    .gnu.hash     : { *(.gnu.hash) }
}

```

里面定义了入口函数是 `_start()`（注意不是 `main()`!!!），以及加载位置是 `0x8000`。其中 `crtbegin.c` 至关重要，`main()` 函数在 `_libc_init()` 里调用到，而之前已经做了很多事，这也就是 `crt`（c-runtime, c 运行环境）的工作。

3. 动态链接

`build/core/combo/TARGET_linux-arm.mk` 里有定义 `transform-o-to-executable-inner` 和 `transform-o-to-shared-lib-inner`，分别将*.o 链接为动态可执行程序 and 共享库。动态可执行程序需要 linker 才能进一步运行的。链接器也是用 `arm-linux-androideabi-g++`，下面分析讲解：

(1). 动态可执行程序

主要的参数和介绍如下：

```

-Bdynamic    ##表示动态链接##
-fPIE        ##链接为位置无关##
-pie
-Wl,-z,now   ##表示立即绑定##
-Wl,-dynamic-linker,/system/bin/linker    ##指定解释器##
-lc -lstdc++ -lm    ##所需的 lib 库，需要显式指定，否则编译报错##
out/target/product/$proj/obj/lib/crtbegin_dynamic.o
out/target/product/$proj/EXECUTABLES/test_intermediates/test.o
prebuilts/gcc/linux-x86/arm/arm-linux-androideabi-$version/lib/gcc/arm-linux-androideabi/$
version-google/armv7-a/libgcc.a
out/target/product/$proj/obj/lib/crtend_android.o

```

相比静态链接，多了 `PIE`，立即绑定，解释器。`crtbegin_dynamic.o` 和 `crtbegin_static.o` 一样。另外的差别是加载地址可以有系统决定，不像静态链接程序固定到 `0x8000`。

(2). 共享库

主要的参数和介绍如下：

```

-Wl,-shared,-Bsymbolic    ##编译为共享库##
-Wl,-z,now                ##表示立即绑定##
-lc -lstdc++ -lm          ##该 lib 所需的 lib 库，需要显式指定，否则编译报错##
out/target/product/$proj/obj/lib/crtbegin_so.o
out/target/product/$proj/EXECUTABLES/test_intermediates/test.o
prebuilts/gcc/linux-x86/arm/arm-linux-androideabi-$version/lib/gcc/arm-linux-androideabi/$
version-google/armv7-a/libgcc.a
out/target/product/$proj/obj/lib/crtend_so.o

```

相比动态可执行程序，没有太大变化，不过 `crtbegin_so.o` 和 `crtend_so.o` 少了 `_start()` 等相关代码，共享库不需要入口函数。

4. 移除调试信息

链接后的程序一般比较大，因为包含了调试信息，而这些信息不会在运行时用到，所以会将其删除后在放入手机里。这里用到了 `arm-linux-androideabi-strip --strip-all` 功能：



5. libgcc.a

GCC 在一些平台上提供了一个低级运行时库，libgcc.a 或者 libgcc_s.so.1。一旦需要执行某些过于复杂而无法通过内嵌代码实现的操作，GCC 便会自动生成对这些库函数的调用。

大多数 libgcc 中的函数用来处理目标处理器不能直接执行的算术运算（包括整数乘除，所有浮点运算），还包括异常处理，少数杂项操作。

详细请参考：<http://gcc.gnu.org/onlinedocs/gccint/Libgcc.html#Libgcc>

6. crt*.o

这些都是 c 运行时库，用于执行进入 main 之前的初始化和退出 main 之后的扫尾工作：

__PREINIT_ARRAY__, __INIT_ARRAY__: 初始化时调用的函数数组

__FINI_ARRAY__: 结束时调用的函数数组

__CTOR_LIST__: 初始化时调用的构造函数数组

这些数组都是以 -1 开始，以 0 结束：

对应的代码 (crtbegin.c) 如下：


```

__attribute__((section(".preinit_array")))
void (*__PREINIT_ARRAY__)(void) = (void (*)(void)) -1;

__attribute__((section(".init_array")))
void (*__INIT_ARRAY__)(void) = (void (*)(void)) -1;

__attribute__((section(".fini_array")))
void (*__FINI_ARRAY__)(void) = (void (*)(void)) -1;

__LIBC_HIDDEN__ void __start() {
    structors_array_t array;
    array.preinit_array = &__PREINIT_ARRAY__;
    array.init_array = &__INIT_ARRAY__;
    array.fini_array = &__FINI_ARRAY__;

    void* raw_args = (void*) ((uintptr_t) __builtin_frame
    __libc_init(raw_args, NULL, &main, &array);
}

```

结语

上面只是讲解编译涉及到的部分知识，实际上用到的技术更多，需要大家自己深入了解和实践。

2. ELF/coredump 结构

历史

ELF-可执行链接格式最初是由 UNIX 系统实验室 (USL) 作为应用程序二进制接口 (ABI) 开发和发行。

工具接口标准委员会 TIS 已经将 ELF 作为运行在 Intel32 位架构之上的各类型操作系统的可导出对象文件格式标准。

ELF 标准为开发者提供了一组横跨多运行环境的二进制接口定义来组织软件开发。

1. 类型

(1). 定义（目标文件格式主要三种）

可重定向文件 (Relocatable file)：文件保存着代码和适当的数据，用来和其他的目标文件一起来创建一个可执行文件或者是一个共享目标文件。由编译器和汇编器生成，将由链接器处理。

可执行文件 (Executable File)：文件保存着一个用来执行的程序；该文件指出了 exec 如何来创建程序进程映像。所有重定向和符号都解析完成了，如果存在共享库的链接，那么将在运行时解析。

共享目标文件 (Shared object file)：就是所谓的共享库。文件保存着代码和合适的数据，用来被下面的两个链接器链接。第一个是连接编辑器 [ld]，可以和其他的可重定向和共享目标文件来创建其他的目标文件。第二个是动态链接器，联合一个可执行文件和其他的共享目标文件来创建一个进程映像。

包含链接时所需的符号信息和运行时所需的代码。



2). 识别

通过 file 命令可以看到哪种类型的 ELF，比如：

```
gcc -c helloworld.c
```

```
file helloworld.o
```

```
helloworld.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped ##可重定向文件
```

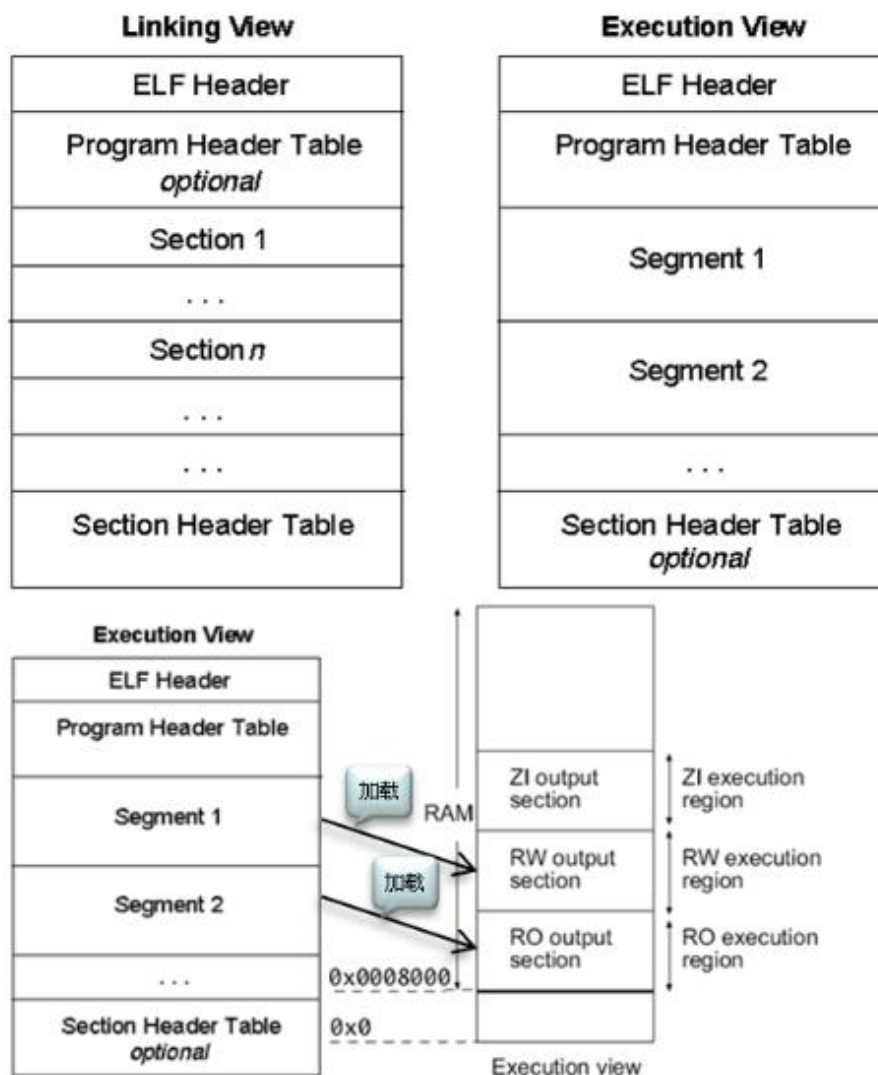
##

```
gcc -o helloworld helloworld.o
file helloworld
helloworld: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (uses shared
libs), for GNU/Linux 2.6.15, not stripped ##可执行文件##
```

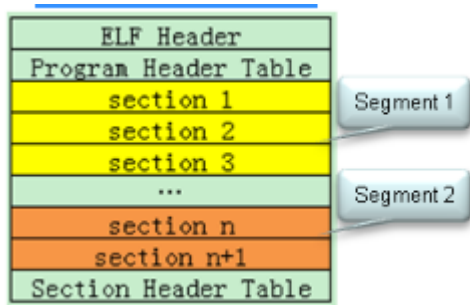
```
file libc.so
libc.so: ELF 32-bit LSB shared object, ARM, version 1 (SYSV), dynamically linked, not stripped ##
共享目标文件##
```

2. 格式

目标文件既要参与程序链接又要参与程序执行。出于方便性和效率考虑，目标文件格式提供了两种并行视图，分别反映了这些活动的不同需求。编译器，链接器把它看作是 sections 的集合，loader 把它看作是 segments 的集合：



段和节是存在包含关系：



节和段描述的数据是一样的

Section	Segment	Sections...
00		
01		.hash .dynsym .dynstr .rel.plt .rel.dyn .plt .text .rodata .ARM.extab .ARM.exidx
02		.init_array .fini_array .data.rel.ro .dynamic .got .data .bss
03		.dynamic
04		
05		.ARM.exidx

该段被加载到RO区,会做MMU保护

其他的段做附加说明

该段被加载到RW/ZI区

3. 结构

(1). 索引表

一般的 ELF 文件包括三个索引表:

ELF header

在文件的开始, 保存了路线图(road map), 描述了该文件的组织情况。

```
/* ELF Header */
typedef struct elfhdr {
    unsigned char e_ident[16]; /* ELF Identification */
    Elf32_Half e_type; /* object file type */
    Elf32_Half e_machine; /* machine */
    Elf32_Word e_version; /* object file version */
    Elf32_Addr e_entry; /* virtual entry point */
    Elf32_Off e_phoff; /* program header table offset */
    Elf32_Off e_shoff; /* section header table offset */
    Elf32_Word e_flags; /* processor-specific flags */
    Elf32_Half e_ehsize; /* ELF header size */
    Elf32_Half e_phentsize; /* program header entry size */
    Elf32_Half e_phnum; /* number of program header entries */
    Elf32_Half e_shentsize; /* section header entry size */
    Elf32_Half e_shnum; /* number of section header entries */
    Elf32_Half e_shstrndx; /* section header table's "section
                             header string table" entry offset */
} Elf32_Ehdr;
```

用 readelf 查看 elf 头部信息如下:

```

[ ELF]$ readelf -h libc.so
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                               ELF32
  Data:                               2's complement, little endian
  Version:                             1 (current)
  OS/ABI:                              UNIX - System V
  ABI Version:                         0
  Type:                                DYN (Shared object file)
  Machine:                             ARM
  Version:                             0x1
  Entry point address:                 0x0
  Start of program headers:            52 (bytes into file)
  Start of section headers:            1414476 (bytes into file)
  Flags:                               0x5000000, Version5 EABI
  Size of this header:                 52 (bytes)
  Size of program headers:             32 (bytes)
  Number of program headers:           6
  Size of section headers:             40 (bytes)
  Number of section headers:           34
  Section header string table index: 33

```

Section header table

包含了描述文件节区的信息，每个节区在表中都有一项，每一项给出诸如节区名称、节区大小这类信息。用于链接的目标文件必须包含节区头部表，其他目标文件可选。

ELF header 中的 `e_shoff` 变量就是保存 section header table 入口对文件头的偏移量。

```

/* Section Header */
typedef struct {
    Elf32_Word sh_name; /* name - index into section header
                        string table section */
    Elf32_Word sh_type; /* type */
    Elf32_Word sh_flags; /* flags */
    Elf32_Addr sh_addr; /* address */
    Elf32_Off sh_offset; /* file offset */
    Elf32_Word sh_size; /* section size */
    Elf32_Word sh_link; /* section header table index link */
    Elf32_Word sh_info; /* extra information */
    Elf32_Word sh_addralign; /* address alignment */
    Elf32_Word sh_entsize; /* section entry size */
} Elf32_Shdr;

```

和该节有
关联的节
的索引

通过 Section header table，我们可以定位到所有的节。

Program header table

告诉系统如何创建进程映像。用来构造进程映像的目标文件必须具有程序头部表，[可重定位文件不需要这个表](#)。

ELF header 中的 `e_phoff` 变量就是保存 program header table 入口对文件头的偏移量。

```

/* Program Header */
typedef struct {
    Elf32_Word p_type; /* segment type */
    Elf32_Off p_offset; /* segment offset */
    Elf32_Addr p_vaddr; /* virtual address of segment */
    Elf32_Addr p_paddr; /* physical address - ignored? */
    Elf32_Word p_filesz; /* number of bytes in file for seg. */
    Elf32_Word p_memsz; /* number of bytes in mem. for seg. */
    Elf32_Word p_flags; /* flags */
    Elf32_Word p_align; /* memory alignment */
} Elf32_Phdr;

```

通过 program header table，我们可以定位到所有的段。

(2). Section

在 ELF spec 里有预定义了几个节结构：

```

/* Section names */
#define ELF_BSS      ".bss"      /* uninitialized data */
#define ELF_DATA     ".data"     /* initialized data */
#define ELF_DEBUG    ".debug"    /* debug */
#define ELF_DYNAMIC   ".dynamic"  /* dynamic linking information */
#define ELF_DYNSTR    ".dynstr"   /* dynamic string table */
#define ELF_DYNSYM    ".dynsym"   /* dynamic symbol table */
#define ELF_FINI      ".fini"     /* termination code */
#define ELF_GOT       ".got"      /* global offset table */
#define ELF_HASH      ".hash"     /* symbol hash table */
#define ELF_INIT      ".init"     /* initialization code */
#define ELF_REL_DATA  ".rel.data"  /* relocation data */
#define ELF_REL_FINI  ".rel.fini"  /* relocation termination code */
#define ELF_REL_INIT  ".rel.init"  /* relocation initialization code */
#define ELF_REL_DYN    ".rel.dyn"  /* relocation dynamic link info */
#define ELF_REL_RODATA ".rel.rodata" /* relocation read-only data */
#define ELF_REL_TEXT  ".rel.text"  /* relocation code */
#define ELF_RODATA    ".rodata"    /* read-only data */
#define ELF_SHSTRTAB   ".shstrtab" /* section header string table */
#define ELF_STRTAB     ".strtab"    /* string table */
#define ELF_SYMTAB     ".symtab"    /* symbol table */
#define ELF_TEXT       ".text"     /* code */

```

这边只介绍下以下节：

a. symtab 和 dynsym 节

用于存储符号名字，值，类型等信息：

```

/* Symbol Table Entry */
typedef struct elf32_sym {
    Elf32_Word st_name; /* name - index into string table */
    Elf32_Addr st_value; /* symbol value */
    Elf32_Word st_size; /* symbol size */
    unsigned char st_info; /* type and binding */
    unsigned char st_other; /* 0 - no defined meaning */
    Elf32_Half st_shndx; /* section header index */
} Elf32_Sym;

```

类型可以表示函数/变量等：

```

/* Symbol type - ELF32_ST_TYPE - st_info */
#define STT_NOTYPE 0 /* not specified */
#define STT_OBJECT 1 /* data object */
#define STT_FUNC 2 /* function */
#define STT_SECTION 3 /* section */
#define STT_FILE 4 /* file */
#define STT_NUM 5 /* number of symbol types */
#define STT_LOPROC 13 /* reserved range for processor */
#define STT_HIPROC 15 /* specific symbol types */

```

使用 readelf 导出符号表：

```

[ ELF]$areadelf -s libc.so|more

Symbol table '.dynsym' contains 1181 entries:
  Num:   Value   Size Type   Bind   Vis      Ndx Name
  0: 00000000    0 NOTYPE  LOCAL  DEFAULT  UND
  1: 00011750   12 FUNC    GLOBAL DEFAULT    7 __get_thread
  2: 0000ff84    8 FUNC    GLOBAL DEFAULT    7 __acabi_unwind_cpp_pr0
  3: 0001175c   28 FUNC    GLOBAL DEFAULT    7 __get_stack_base
  4: 00011778   48 FUNC    GLOBAL DEFAULT    7 pthread_attr_init

```

b. init/fini/ctors 节

在讲 crt 章节有介绍到。preinit_array, .init_array, .fini_array, .ctors 节结构是一样的，都是函数指针数组（开始于-1，结束于 0）

Section Headers:										
[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[10]	.preinit_array	PREINIT_ARRAY	00009000	001000	000008	00	WA	0	0	1
[11]	.init_array	INIT_ARRAY	00009008	001008	000008	00	WA	0	0	1
[12]	.fini_array	FINI_ARRAY	00009010	001010	000008	00	WA	0	0	1
[13]	.ctors	PROGBITS	00009018	001018	000008	00	WA	0	0	1
.CTOR_LIST__							D:0			
data_start							C:0			
dso_handle							D:0			
end__							C:0			
exidx_end							C:0			
exidx_start							C:0			
FINI_ARRAY__							C:0			
INIT_ARRAY__							C:0			
libc_init							C:0			
PREINIT_ARRAY__							C:0			

address	offset	size	comment
NSD:00009000	00000000	00000008	00000179
NSD:00009010	00000000	00000008	00000179
NSD:00009020	00000000	00000008	00000179
NSD:00009030	00000000	00000008	00000179

如何将一个函数添加到这些节当中呢？你可以这样：

```
#define __INIT_ARRAY__ attribute__((constructor))
void __INIT_ARRAY__ before_main()
,
#define __FINI_ARRAY__ attribute__((destructor))
void __FINI_ARRAY__ after_main()
,
```

添加到 .init_array 的函数将在 main() 函数前就开始执行了，在 bionic 有些函数使用了这样的手段，大家可以将它们找出来。

全局 C++ 类的构造函数也会被添加到 __INIT_ARRAY__ 中。编译器会额外生成一个函数，该函数将文件中的所有全局类调用构造函数，并将析构函数注册到 atexit 函数。

c. Debug 节

采用 DWARF 标准存储调试信息，我们分析 NE 都需要这些调试信息，这些信息会被 strip 掉，因此手机里的程序/库的 ELF 没有这部分信息，只有 symbols 目录下的才有（所以一定要保留 symbols 目录！）。

debug 节有很多种类，比如：

- .debug_line：存放代码行号和地址对应信息。
- .debug_frame：存放解栈信息。
- .debug_loc：存放函数里的寄存器代表符号的信息。
- .debug_pubnames：存放全局变量及函数。

这些节在调试中怎么体现出来呢？使用 gdb/trace32，你将看到这些节带来的信息：

addr/line	code	label	mnemonic	comment
NSR:400E3BBC	E3A04000		mov r4, #0x0	: r4, #0
NSR:400E3BC0	E51EC052		strb r12, [r14, #0x52]	
debug_line 4483				
NSR:400E3BC4	EBFFCAE2		bl 0x400D6754	
NSR:400E3BC8	E3A01006		mov r1, #0x6	: p, #6
NSR:400E3BCC	EBFFCF37		bl 0x400D78B0	
NSR:400E3BD0	E28D1004		add r1, r13, #0x4	: p, r13, #4

symbol	type	address
__ettimeofday		P:400D7154-4
__etuid		P:400D65E4-4
__etusershell		P:4010A71C-4
__etusershell		P:4010C800-4
__etusershell		P:4001BC00-4
__etusershell		P:400ECAC4-4
__etusershell		P:400ED300-4
__etusershell		P:40021370-4
__etusershell		P:400CF004-4

(3). segment

段也有不同的类型，有些段中保存着机器指令，有些保存着已初始化的变量，有些则作为进程镜像的一部分被操作系统读入内存。

类型如下：

```
/* Segment types - p_type */
#define PT_NULL 0 /* unused */
#define PT_LOAD 1 /* loadable segment */
#define PT_DYNAMIC 2 /* dynamic linking section */
#define PT_INTERP 3 /* the RTLD */
#define PT_NOTE 4 /* auxiliary information */
#define PT_SHLIB 5 /* reserved - purpose undefined */
#define PT_PHDR 6 /* program header */
#define PT_NUM 7 /* Number of segment types */
#define PT_LOOS 0x60000000 /* reserved range for OS */
#define PT_HIOS 0x6fffffff /* specific segment types */
#define PT_LOPROC 0x70000000 /* reserved range for processor */
#define PT_HIPROC 0x7fffffff /* specific segment types */

/* Segment flags - p_flags */
#define PF_X 0x1 /* Executable */
#define PF_W 0x2 /* Writable */
#define PF_R 0x4 /* Readable */
#define PF_MASKPROC 0xf0000000 /* reserved bits for processor */
```

一般一个完整的共享库或可执行程序存在 2 个 PT_LOAD（该类型会被加载到内存），用于保存代码段和数据段。

```
[ ELF]$readelf -l libc.so

Elf file type is DYN (Shared object file)
Entry point 0x0
There are 6 program headers, starting at offset 52

Program Headers:
  Type           Offset             VirtAddr           PhysAddr          FileSiz MemSiz  Flg Align
  PHDR           0x00000034          0x00000034          0x00000034          0x000c0 0x000c0 R   0x4
  LOAD           0x00000000          0x00000000          0x00000000          0x4258c 0x4258c R E 0x1000
  LOAD           0x043000          0x00043000          0x00043000          0x0dc18 0x0dc18 RW 0x1000
  DYNAMIC        0x0443e0          0x000443e0          0x000443e0          0x000b8 0x000b8 RW 0x4
  GNU_STACK     0x000000          0x00000000          0x00000000          0x00000 0x00000 RW 0
  EXIDX         0x040d4c          0x00040d4c          0x00040d4c          0x01840 0x01840 R   0x4

Section to Segment mapping:
Segment Sections...
00
01  .hash .dynsym .dynstr .rel.plt .rel.dyn .plt .text .rodata .ARM.exidx
   .ARM.exidx
02  .init_array .fini_array .data.rel.ro .dynamic .got .data .bss
03  .dynamic
04
05  .ARM.exidx
```

4. coredump 结构

coredump 只是其中一种 ELF，在经过前面的学习，应该很容易理解 coredump 结构。

coredump 是 ELF 第 4 种结构，仅为调试而存在：

```
/* define the different elf file types
#define ET_NONE 0
#define ET_REL 1
#define ET_EXEC 2
#define ET_DYN 3
#define ET_CORE 4
```

coredump 没有节，只有段，而且只有两种段类型，PT_NOTE 和 PT_LOAD，note 段保存了现场寄存器，线程和引起崩溃信号的信息，而 PT_LOAD 这是进程空间内存数据段。

```

ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                           ELF32
  Data:                             2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                               CORE (Core File)
  Machine:                           ARM
  Version:                           0x1
  Entry point address:                0x0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          0 (bytes into file)
  Flags:                             0x0
  Size of this header:                52 (bytes)
  Size of program headers:            32 (bytes)
  Number of program headers:          18
  Size of section headers:            0 (bytes)
  Number of section headers:          0
  Section header string table index: 0

Program Headers:
  Type Offset VirtAddr FileSiz Flg Align
  NOTE 0x000274 0x00000000 0x0037c 0
# LOAD 0x001000 0x00008000 0x01000 R EP 1f:0b 333 /system/bin/test
  LOAD 0x001000 0x00009000 0x01000 RW P 1f:0b 333 /system/bin/test
  LOAD 0x002000 0x01fa4000 0x02000 RW P 00:00 0 [heap]
  LOAD 0x004000 0x40002000 0x15000 R EP 1f:0b 606 /system/lib/libm.so
  LOAD 0x019000 0x40017000 0x01000 RW P 1f:0b 606 /system/lib/libm.so
# LOAD 0x01a000 0x40028000 0x0d000 R S 00:0b 1398 dev/__properties__ (deleted)
  LOAD 0x01a000 0x40035000 0x01000 R P 00:00 0
  LOAD 0x01b000 0x400c0000 0x01000 R EP 1f:0b 604 /system/lib/libstdc++.so
  LOAD 0x01c000 0x400c1000 0x01000 RW P 1f:0b 604 /system/lib/libstdc++.so
  LOAD 0x01d000 0x400d5000 0x43000 R EP 1f:0b 597 /system/lib/libc.so
  LOAD 0x060000 0x40118000 0x03000 RW P 1f:0b 597 /system/lib/libc.so
  LOAD 0x063000 0x4011b000 0x0b000 RW P 00:00 0 /system/lib/libc.so <= bss
# LOAD 0x06e000 0xb0001000 0x0b000 R EP 1f:0b 318 /system/bin/linker
  LOAD 0x06e000 0xb000c000 0x01000 RW P 1f:0b 318 /system/bin/linker
  LOAD 0x06f000 0xb000d000 0x11000 RW P 00:00 0 /system/bin/linker <= bss
  LOAD 0x080000 0xbef95000 0x22000 RW P 00:00 0 [stack]
  LOAD 0x0a2000 0xfffff000 0x01000 R EP 00:00 0 [vectors]

```

PT_NOTE 结构如下:



其中的 type 可以是:

```

Architectures export some of the arch register sets
using the corresponding note types via the
PTRACE_GETREGSET and PTRACE_SETREGSET requests.
#define NT_PRSTATUS 1
#define NT_PRFPREG 2
#define NT_PRPSINFO 3
#define NT_TASKSTRUCT 4
#define NT_AUXV 6
#define NT_ARM_VFP 0x400 /* ARM VFP/NEON regist
其中NT_PRSTATUS就记录了CPU寄存器信息。

```

其中 NT_PRSTATUS 就记录了 CPU 寄存器信息。

结语

以上只是抛砖引玉，真正全面的资料大家还需参考 ELF 标准和 ARM 对 ELF 的扩展。

3. ELF 加载执行

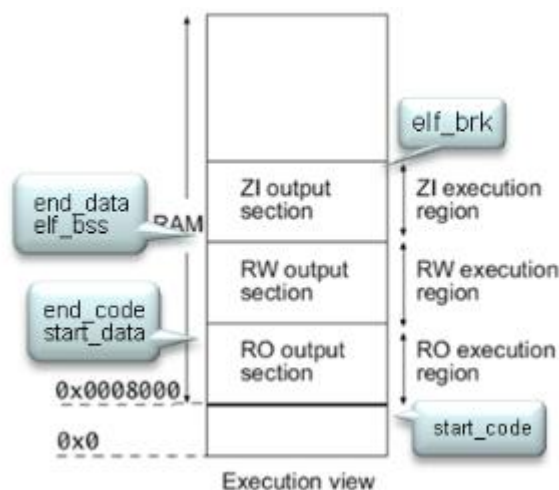
1. ELF loader

这个 ELF loader 指的是 kernel elf loader。可执行程序以 ELF 文件的形式保存在文件系统中，核心的 `load_elf_binary` 会首先将其映像文件映射到内存，然后映射并执行其解释器(linker)的代码。linker 的代码段是进程间共享的，但数据段为各进程私有，linker 执行完后会自动跳转到目标映像的入口地址：

`sys_execve() => do_execve() => search_binary_handler() => load_elf_binary()`

(1). 加载

在 `load_elf_binary()` 函数里，会解析这个程序的 ELF 结构，判断是否为 ELF，然后根据 PT_LOAD 段加载到对应的进程空间内：



如果需要 linker，linker 也会被系统加载。

```
if (elf_interpreter)
{
    elf_entry = load_elf_interp(&loc->interp_elf_ex,
        interpreter, &interp_map_addr, 0);
    /* load_elf_interp() returns relocation adjustment */
    interp_load_addr = elf_entry;
    elf_entry += loc->interp_elf_ex.e_entry;
    allow_write_access(interpreter);
    fput(interpreter);
    kfree(elf_interpreter);
}
else
{
    elf_entry = loc->elf_ex.e_entry;
}
vectors_user_mapping();
```

装载解释器,和之前的步骤一样

入口地址为解释器入口地址

否则为程序的入口地址

(2). 初始化栈

分配好栈后，将参数压入栈中，然后是环境变量，最后是辅助向量。

地址	内容	大小	说明
	eve	4	ccx函数局部变量
	envp	4	main函数局部变量
	auxv	4	
	i	4	
init SP ->	argc (n)	4	参数个数
	argv[0] (pointer)	4	
	argv[1] (pointer)	4	
	argv[...] (pointer)	4	
	argv[n] (pointer)	4	=NULL
	envp[0] (pointer)	4	
	envp[1] (pointer)	4	
	envp[...] (pointer)	4	
	envp[n] (pointer)	4	=NULL
	auxv[0] (Elf32_auxv_t)	8	
	auxv[1] (Elf32_auxv_t)	8	
	auxv[...] (Elf32_auxv_t)	8	
	auxv[x] (Elf32_auxv_t)	8	=AT_NULL
	padding 0	0~15	
	argument 0 strings	>=0	
	argument 1 strings	>=0	
	argument ... strings	>=0	
	argument n strings	>=0	
	environment 0 strings	>=0	
	environment 1 strings	>=0	
	environment ... strings	>=0	
	environment n strings	>=0	
0xBFFFFFFC	end marker	4	=NULL
0xC0000000	<bottom of stack>		

这里介绍下辅助向量，辅助向量会传递内核一些信息，比如CPU型号和功能，页面大小等。

结构如下：

```
typedef struct
{
    uint32_t tag;
    uint32_t value;
} Elf32_auxv_t;
```

kernel/include/linux/auxvec.h

```
/* Symbolic values for the entries in the auxiliary table
   put on the initial stack */
#define AT_NULL 0 /* end of vector */
#define AT_IGNORE 1 /* entry should be ignored */
#define AT_EXECFD 2 /* file descriptor of program */
#define AT_PHDR 3 /* program headers for program */
#define AT_PHEMT 4 /* size of program header entry */
#define AT_PHNUM 5 /* number of program headers */
#define AT_PAGESZ 6 /* system page size */
#define AT_BASE 7 /* base address of interpreter */
#define AT_FLAGS 8 /* flags */
#define AT_ENTRY 9 /* entry point of program */
#define AT_NOTELF 10 /* program is not ELF */
#define AT_UID 11 /* real uid */
#define AT_EUID 12 /* effective uid */
#define AT_GID 13 /* real gid */
#define AT_EGID 14 /* effective gid */
#define AT_PLATFORM 15 /* string identifying CPU for optimizations */
#define AT_HWCAP 16 /* arch dependent hints at CPU capabilities */
#define AT_CLKTCK 17 /* frequency at which times() increments */
/* AT_* values 18 through 22 are reserved */
#define AT_SECURE 23 /* secure mode boolean */
#define AT_BASE_PLATFORM 24 /* string identifying real platform, may
   * differ from AT_PLATFORM. */
#define AT_RANDOM 25 /* address of 16 random bytes */
#define AT_EXECFN 31 /* filename of program */
```

辅助向量

(3). 准备调度

做完之后，就可以等待调度了。

2. linker

Kernel 加载完可执行程序和对应的连接器后，就跳转到用户空间的入口地址 (ICS/GB:0xB0001000, JB 之后:系统动态分配)开始执行。

Android 的加载/链接器 linker 主要用于实现共享库的加载与链接（支持应用程序对库函数的隐式和显式调用）：

隐式调用

应用程序的编译与静态库大致相同，只是在静态链接的时候通过--dynamic-linker /system/bin/linker 指定动态链接器（该信息将被存放在 ELF 文件的.interp 节中，内核执行目标映像文件前将通过该信息加载并运行相应的解释器程序 linker）并链接相应的共享库。

与 ld.so 不同的是，Linker 目前没有提供 Lazy Binding 机制，所有外部过程引用都在映像执行之前解析。

显式调用

可以通过 linker 中提供的接口 dlopen(), dlsym(), dlerror() 和 dlclose() 来动态加载和链接共享库。

linker 里面涉及的加载和其他平台大同小异，大家可以查看相关书籍。

编程篇: linux c 编程

1. 信号

1. 定义

简而言之，信号是一种软件中断，提供了一种处理异步的方法，信号发生是随机的。例如键盘输入中断按键(C)，它的发生在程序执行过程中是不可预测的。

硬件异常也能产生信号，例如被零除、无效内存引用（test 里产生的就是这种错误）等。这些条件通常先由内核硬件检测到，然后通知内核。内核将决定产生什么样的信号。

同一个信号的额外发生通常不会被排队。如果信号在被阻塞时发生了 5 次，当我们反阻塞这个信号时，这个信号的信号处理函数通常只被调用一次。

同一时刻只能处理一个信号，在信号处理函数发信号给自己时，该信号会被 pending。

信号的数值越小，则优先级越高。当进程收到多个待处理信号时，总是先处理优先级高的信号。

信号处理函数的栈可以使用被中断的也可以使用独立的，具体可以通过系统调用设置。

2. 处理方式

忽略：接收到信号后不做任何反应。

捕获：用自定义的信号处理函数来执行特定的动作。

默认：接收到信号后按系统默认的行为处理该信号。

这是多数应用采取的处理方式

3. 类型

这里列出主要的信号，具体可查看 bionic/libc/kernel/arch-arm/asm/signal.h:

名称	数字	标准	默认行为	说明
SIGILL	4	ANSI	终止 +coredump	执行了非法指令。通常是因为可执行文件本身出现错误，或者试图执行数据段。堆栈溢出时也有可能产生这个信号
SIGABRT	6	ANSI	终止 +coredump	调用 abort 函数生成的信号

SIGBUS	7	4.2 BSD	终止 +coredump	非法地址，包括内存地址对齐(alignment)出错。比如访问一个四个字长的整数，但其地址不是4的倍数。它与 SIGSEGV 的区别在于后者是由于对合法存储地址的非法访问触发的(如访问不属于自己存储空间或只读存储空间)
SIGFPE	8	ANSI	终止 +coredump	在发生致命的算术运算错误时发出。不仅包括浮点运算错误，还包括溢出及除数为0等其它所有的算术的错误
SIGSEGV	11	ANSI	终止 +coredump	试图访问未分配给自己的内存，或试图往没有写权限的内存地址写数据。访问空指针，野指针基本都产生这个信号，也是最常见信号
SIGSTKFLT	16	N/A	终止	堆栈错误
SIGPIPE	13	POSIX	终止	管道破裂。这个信号通常在进程间通信产生，比如采用 FIFO(管道)通信的两个进程，读管道没打开或者意外终止就往管道写，写进程会收到 SIGPIPE 信号。此外用 Socket 通信的两个进程，写进程在写 Socket 的时候，读进程已经终止
SIGTRAP	5	POSIX	终止 +coredump	由断点指令或其它 trap 指令产生。由 debugger 使用
SIGHUP	1	POSIX	终止	用户终端连接(正常或非正常)结束时发出，通常是在终端的控制进程结束时，通知同一 session 内的各个作业，这时它们与控制终端不再关联
SIGINT	2	ANSI	终止	程序终止(interrupt)信号，在用户键入 INTR 字符(通常是 Ctrl-C)时发出，用于通知前台进程组终止进程
SIGQUIT	3	POSIX	终止 +coredump	和 SIGINT 类似，但由 QUIT 字符(通常是 Ctrl-\)来控制。进程在因收到 SIGQUIT 退出时会产生 core 文件，在这个意义上类似于一个程序错误信号
SIGKILL	9	POSIX	终止	用来立即结束程序的运行。本信号不能被阻塞、捕获和忽略。如果管理员发现某个进程终止不了，可尝试发送这个信号
SIGCHLD	17	POSIX	忽略	子进程结束时，父进程会收到这个信号。如果父进程没有处理这个信号，也没有等待(wait)子进程，子进程虽然终止，但是还会在内核进程表中占有表项，这时的子进程称为僵尸进程。这种情况我们应该避免(父进程或者忽略 SIGCHLD 信号，或者捕捉它，或者 wait 它派生的子进程，或者父进程先终止，这时子进程的终止自动由 init 进程来接管)
SIGCONT	18	POSIX	继续/忽略	让一个停止(stopped)的进程继续执行。本信号不能被阻塞。可以用一个 handler 来让程序在由 stopped 状态变为继续执行时完成特定的工作。例如，重新显示提示符。在进程挂起时是继续，否则是忽略
SIGSTOP	19	POSIX	暂停	暂停进程的执行。注意它和 terminate 以及 interrupt 的区别:该进程还未结束，只是暂停执行。本信号不能被阻塞、捕获或忽略
SIGALRM	14	POSIX	终止	时钟定时信号，计算的是实际的时间或时钟时间。alarm 函数使用该信号

【TIPS】：编号为0的信号，用以测试进程是否拥有信号发送的权限，并不会被实际发送

4. 详细信息

kernel 可以传递更多信号相关的信息给 native 层，通过 ptrace 或 waitid 函数可以获取信号详细信息。

信息保存在 siginfo 结构体，如下：

```

typedef struct siginfo
{
    int si_signo; /* 信号 */
    int si_errno; /* 错误码 */
    int si_code; /* 额外的信息，基于信号 */
    union {
        struct { /* SIGILL, SIGFPE, SIGSEGV, SIGBUS */
            void * _user = _addr; /* faulting insn/memory ref. */
            short _addr_lsb; /* LSB of the reported address */
        } _sigfault;
        .....
    } _sifields;
} siginfo_t;

```

下表展示了各种信号的 si_code 值，由 SUS(Single UNIX Specification:是 POSIX.1 标准的超集)定义：

SIGSEGV 的代码和原因：

代码	原因
SEGV_MAPERR: 1	地址没有映射到对象（大部分的异常是这种类型）
SEGV_ACCERR: 2	映射的对象的无效权限

SIGBUS 的代码和原因：

代码	原因
BUS_ADRALN: 1	地址不对齐(自然对齐)
BUS_ADRERR: 2	不存在的物理地址
BUS_OBJERR: 3	对象指定的硬件错误

SIGFPE 的代码和原因：

代码	原因
FPE_INTDIV: 1	整数被 0 除
FPE_INTOVF: 2	整数溢出
FPE_FLDIV: 3	浮点数被 0 除
FPE_FLOVF: 4	浮点数溢出
FPE_FLTUND: 5	浮点数下溢
FPE_FLTRES: 6	浮点数不精确结果
FPE_FLTINV: 7	无效的浮点数操作
FPE_FLTSUB: 8	范围外的下标

SIGILL 的代码和原因：

代码	原因
ILL_ILLOPC: 1	违法操作码
ILL_ILLOPN: 2	违法操作数
ILL_ILLADR: 3	违法地址模式
ILL_ILLTRP: 4	违法陷阱
ILL_PRVOPC: 5	特权操作码
ILL_PRIVREG: 6	特权寄存器
ILL_COPROC: 7	协进程错误
ILL_BADSTK: 8	内部栈错误

SIGTRAP 的代码和原因：

代码	原因
TRAP_BRKPT: 1	进程中断点陷阱
TRAP_TRACE: 2	进程跟踪陷阱

SIGCHLD 的代码和原因:

代码	原因
CLD_EXITED: 3	子进程已经退出
CLD_KILLED: 4	子进程已异常退出(无 coredump)
CLD_DUMPED: 5	子进程已异常退出(有 coredump)
CLD_RAPPED: 6	跟踪的子进程已经被套住
CLD_STOPPED: 7	子进程被停止
CLD_CONTINUED: 8	停止的子进程被继续

SIGPOLL 的代码和原因:

代码	原因
POLL_IN: 1	数据可以被读
POLL_OUT: 2	数据可以被写
POLL_MSG: 3	输入消息可用
POLL_ERR: 4	I / O 错误
POLL_PRI: 5	高优先级消息可用
POLL_HUP: 6	设备断开连接

2. socket/ptrace

3. 1. socket

(1). 介绍

使用套接字除了可以实现网络间不同主机间的通信外, 还可以实现同一主机的不同进程间的通信, 且建立的通信是双向的通信。

(2). 服务器端建立 (socket -> bind -> listen -> accept -> read/write)

程序通过调用 `socket` 函数 (AF_LOCAL, SOCK_STREAM), 建立了主动连接的套接字。

调用 `bind` 函数, 将套接字与地址信息关联起来。

调用 `listen` 函数实现对该端口的监听, 同时变为监听套接字。

当有连接请求时, 通过调用 `accept` 函数建立与客户端的连接。

调用 `read/write` 函数来读取/发送消息, 当然也可以使用 `recv/send` 函数实现相同的功能。

(3). 客户端建立 (socket -> connect -> read/write)

程序通过调用 `socket` 函数 (AF_LOCAL, SOCK_STREAM) , 建立了主动连接的套接字。

调用 `connect` 函数(附带地址信息), 向服务器端发出连接请求。

调用 `read/write` 函数来读取/发送消息, 当然也可以使用 `recv/send` 函数实现相同的功能。

2. ptrace (process trace)

(1). 介绍

`ptrace()` 是个系统调用, 提供了一种父进程可以控制子进程运行, 并可以检查和改变它的核心 image (用于实现断点调试, 代码分析)

修改被跟踪进程的空间(内存或寄存器)。

任何传递给被跟踪进程的信号(除了 SIGKILL) 都会使得这个进程进入暂停状态,这时跟踪进程通过 wait() 得知相关的状态并做相应的修改。

4. (2). 限制

不能跟踪进程 init。不能跟踪自己。

(3). 功能

#define PTRACE_TRACEME	0x00	控制类
#define PTRACE_CONT	0x07	
#define PTRACE_KILL	0x08	
#define PTRACE_SINGLESTEP	0x09	
#define PTRACE_ATTACH	0x10	
#define PTRACE_DETACH	0x11	
#define PTRACE_SYSCALL	0x18	
#define PTRACE_GETEVENTMSG	0x4201	信息读取/设置
#define PTRACE_OLDSETOPTIONS	0x15	
#define PTRACE_SETOPTIONS	0x4200	
#define PTRACE_GET_THREAD_AREA	0x16	
#define PTRACE_SET_SYSCALL	0x17	
#define PTRACE_GETSIGINFO	0x4202	
#define PTRACE_SETSIGINFO	0x4203	
#define PTRACE_PEEKTEXT	0x01	内存读取/设置
#define PTRACE_PEEKDATA	0x02	
#define PTRACE_POKETEXT	0x04	
#define PTRACE_POKEDATA	0x05	
#define PTRACE_PEEKUSR	0x03	
#define PTRACE_POKEUSR	0x06	
#define PTRACE_POKEUSER	PTRACE_POKEUSR	
#define PTRACE_PEEKUSER	PTRACE_PEEKUSR	
#define PTRACE_GETREGS	0x0C	寄存器读取/设置
#define PTRACE_SETREGS	0x0D	
#define PTRACE_GETFPREGS	0x0E	
#define PTRACE_SETFPREGS	0x0F	
#define PTRACE_GETVFPREGS	0x1B	
#define PTRACE_SETVFPREGS	0x1C	

PTRACE_TRACEME

本进程被其父进程所跟踪。其父进程应该希望跟踪子进程,用于 debugger。

PTRACE_ATTACH

跟踪指定 pid 进程,成为 pid 的父进程,并停止 pid 进程。

PTRACE_DETACH

结束跟踪。

PTRACE_PEEKTEXT, PTRACE_PEEKDATA 和 PTRACE_POKETEXT, PTRACE_POKEDATA

读取/修改被跟踪进程的 user space 里的内存。

PTRACE_GETREGS, PTRACE_SETREGS、PTRACE_GETFPREGS, PTRACE_SETFPREGS、PTRACE_GETVFPREGS,

PTRACE_SETVFPREGS

读取/修改被跟踪进程的通用/浮点寄存器值。

PTRACE_SYSCALL, PTRACE_CONT

重新运行, (PTRACE_SYSCALL 会使每次系统调用暂停)。

PTRACE_SINGLESTEP

设置单步执行标志。

PTRACE_KILL

杀掉子进程,使它退出。

4. libc 功能模块

1. libc

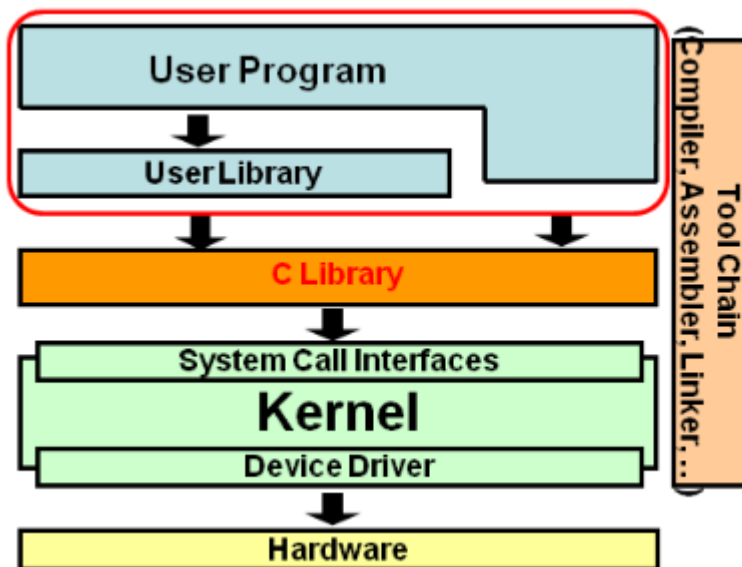
- (1). libc 是 Standard C library 的简称，它是符合 ANSI C 标准的一个函数库。

libc 库提供 C 语言中所使用的宏，类型定义，字符串操作函数，数学计算函数以及输入输出函数等。

正如 ANSI C 是 C 语言的标准一样，libc 只是一种函数库标准，每个操作系统都会按照该标准对标准库进行具体实现。

通常我们所说的 libc 是特指某个操作系统的标准库，比如我们在 Linux 操作系统下所说的 libc 即 glibc。glibc 是类 Unix 操作系统中使用最广泛的 libc 库，它的全称是 GNU C Library。

- (2). 类 Unix 操作系统通常将 libc 库作为操作系统的一部分（被视为操作系统与用户程序之间的接口）



libc 库不仅实现标准 C 语言中的函数，而且也包含自己所属的函数接口。比如在 glibc 库中，既包含标准 C 中的 `fopen()`，又包含类 Unix 系统中的 `open()`。在类 Unix 操作系统中，如果缺失了标准库，那么整个操作系统将不能正常运转。在 Android 也是一样的：



而 Windows 系统并不将 libc 库作为整个核心操作系统的一部分。通常每个编译器都附属自己的 libc 库，这些 libc 既可以静态编译到程序中，又可以动态编译到程序中。也就是说应用程序依赖编译器而不是操作系统。

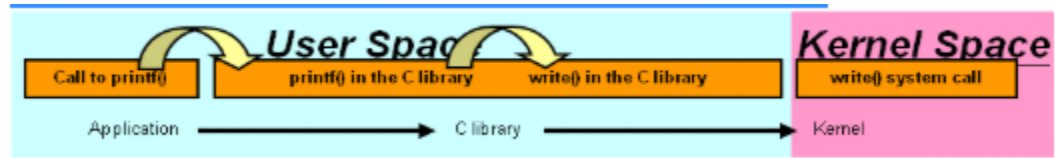
(3). 封装函数

在 Linux 系统中, glibc 库中包含许多 API, 大多数 API 都对应一个系统调用, 比如应用程序中使用的接口 `open()` 就对应同名的系统调用 `open()`。

glibc 库中通过封装例程(Wrapper Routine)将 API 和系统调用关联起来。

API 是头文件中所定义的函数接口, 而位于 glibc 中的封装例程则是对该 API 对应功能的具体实现。

事实上接口 `open()` 所要完成的功能是通过系统调用 `open()` 完成的, 因此封装例程要做的工作是先将接口 `open()` 中的参数复制到相应寄存器中, 然后引发一个异常, 从而系统进入内核去执行 `sys_open()`, 最后当系统调用执行完毕后, 封装例程还要将错误码返回到应用程序中。



注意】 函数库中的 API 和系统调用并没有一一对应的关系。应用程序借助系统调用可以获得内核所提供的服务, 像字符串操作这样的函数并不需要借助内核来实现, 因此也就不必与某个系统调用关联。

也不是必须通过封装例程才能使用系统调用, `syscall()` 和 `_syscallx()` 两个函数可以直接调用系统调用。

2. pthread

bionic libc 包含了 pthread, pthread 另有标准定义, 大家也可以参考。

3. dlmalloc

KK 及以前版本用的是 dlmalloc 作为 malloc/free 的分配器, 学习 dlmalloc 对于堆内存管理是非常必要的。具体资料可以到网络搜索。

结语

libc 是 linux/Android 最基础的库, 掌握它将更好理解 NE。

专题篇：专题分析

1. 踩内存专题

踩内存问题一直都困扰着每个 Android 工程师, 调试难度很大, 处于项目的各个阶段, 严重影响软件品质。我们专门列一个专题分析, 详情请看:

- MediaTek On-Line> Quick Start> 踩内存专题分析

2. fd 专题

fd 泄漏和意外关闭 fd 引起的逻辑问题在 android 也是比较常见的问题, 经过几代的发展, 调试已相当方便, 已经形成固定的套路分析了, 详情请看:

- MediaTek On-Line> Quick Start> 文件描述符(fd)专题分析

通过这篇专题文章, 相信这类问题大家就可以自行处理掉了。

3. 售后收集重启专题

售后软件品质已越来越重要，以前通过返修/客退机来分析解决量产前未发现的问题，现在有更好的方法，通过后台自动收集故障信息，然后回传处理分析。我们专门列一个专题分析，详情请看：

- [MediaTek On-Line> Quick Start> 售后收集重启专题分析](#)

实例篇：案例分析

1. 移除 SD 卡后引发 SIGBUS NE

问题背景：

移除 SD 卡，打開 contacts 時 100% 發生 com.android.contacts 的 Native Exception

分析过程：

使用 GAT 解析 DB 文件，从_exp_main.txt 看到 exception type 是 SIGBUS，对于 SIGBUS 类型的 NE 最常见的一种情况是后备存储器异常（不见了等）导致，比如文件映射进来，结果访问时，文件被删除了。

_exp_main.txt：

```
Exception Class: Native (NE)
Exception Type: SIGBUS

Current Executing Process:
pid: 8016, tid: 8033
com.android.contacts
```

从_exp_detail.txt 可查看到具体的 fault address：

```
pid: 8016, tid: 8033, name: AccountChangeLi >>> com.android.contacts <<<
signal 7 (SIGBUS), code 2 (BUS_ADRERR), fault addr 0x7f910c4000
```

从 PROCESS_MAPS 文件可知 fault address 落在：

```
7f910c3000-7f910c5000 r--s 02929000 fd:01 12 /mnt/asec/com.facebook.katana-1/base.apk
```

此目录/mnt 是 SD 卡的路径。

根本原因：

移除 SD 卡的行为正是此题 NE 发生的原因，com.android.contacts 需要访问安装在 SD 卡上的应用，而此时 SD 卡已经被拔了。

解决方法:

应用不建议安装在 SD 卡上, 如果是安装在 SD 卡, 请不要随意插拔 SD 卡。

2. 孤儿进程组问题导致的重启

问题背景:

- 1、安装豌豆荚, QQ 后
- 2、用豌豆荚下载搜狗输入法, 天天德州, 天天爱消除, 泡泡龙亚特, 并运行这些应用
- 3、点击豌豆荚时, 手机重启

分析过程:

1. 查看 kernel log 看到:

```
<7>[ 1884.162500] -(1)[11342:libxguardian.so][name:mtprof&][signal][11342:libxguardian.so] send death sig 1 to
[1094:ndroid.systemui:W]
<7>[ 1884.162984] -(1)[11342:libxguardian.so][name:mtprof&][signal][11342:libxguardian.so] send death sig 1 to
[988:system_server:W]
<7>[ 1884.163024] -(1)[11342:libxguardian.so][name:mtprof&][signal][11342:libxguardian.so] send death sig 1 to
[258:main:W]
<13>[ 1884.166606] (2)[1:init]init: Untracked pid 11342 exited with status 0
<7>[ 1884.167235] -(2)[11212:libuuid.so][name:mtprof&][signal][11212:libuuid.so] send death sig 6 to
[11212:libuuid.so:R]
<12>[ 1884.290928] (1)[11335:logd.reader.per]logd: logd.reader.per thread stop.
<13>[ 1884.313276] (3)[1:init]init: Untracked pid 11280 killed by signal 1
<13>[ 1884.315574] (1)[1:init]init: Service 'zygote' (pid 258) killed by signal 1
<13>[ 1884.315630] (1)[1:init]init: Service 'zygote' (pid 258) killing any children in process group
system_server 和 zygote 被杀导致 android reboot
```

从以上这段 log 可以看出是孤儿进程组的问题:

若父进程退出导致进程组成为孤儿进程组, 且该进程组中有进程处于停止状态(收到 SIGSTOP 或 SIGTSTP 信号), SIGHUP 信号会被发送到该进程组中的每一个进程。

系统对 SIGHUP 信号的默认处理是终止收到该信号的进程。所以若程序中没有捕捉该信号, 当收到该信号时, 进程就会退出。

孤儿进程组条件:

- App fork child process.
- App exit.
- Child process exit and there is a(or more) process is stop.

关于孤儿进程组的问题也可以参考 FAQ:

[\[FAQ11577\]Linux 孤儿进程组问题说明](#)

此笔 case 中, libxguardian.so 是孤儿进程, 而同在这个孤儿进程组中的 libuuid.so 因为 NE 而处于 stop 状态, 所以 SIGHUP 信号会被发送到该进程组中的每一个进程, 包括 zygote, 所以 android 重启了。

2. 用 GAT 解析 libuuid.so(豌豆荚) NE 的 DB

Exception Class: Native (NE)

Exception Type: SIGABRT

backtrace:

```
#00 pc 00045464 /system/bin/linker (__dl_tgkill+12)

#01 pc 00044a1b /system/bin/linker (__dl_pthread_kill+34)

#02 pc 0003cdbf /system/bin/linker (__dl_raise+10)

#03 pc 0003b86b /system/bin/linker (__dl___libc_android_abort+34)

#04 pc 0003a6c0 /system/bin/linker (__dl_abort+4)

#05 pc 0003c8bb /system/bin/linker (__dl___libc_fatal+22)

#06 pc 000093bb /system/bin/linker (__dl_ZL29__linker_init_post_relocationR19KernelArgumentBlockj+2250)

#07 pc 00008a5b /system/bin/linker (__dl__linker_init+358)

#08 pc 00002770 /system/bin/linker (_start+4)
```

通过 addr2line 定位到在以下函数中调用的 abort:

linker.cpp 中的

```
4258 if (elf_hdr->e_type != ET_DYN) {

4259     __libc_fatal("%s\\": error: only position independent executables (PIE) are supported.",

4260                 args.argv[0]);

4261 }
```

在 NE 之前的 main log 中也看到:

```
F libc : "/data/user/0/com.wandoujia.phoenix2/lib/libuuid.so": error: only position independent executables
(PIE) are supported.
```

异常的应用不是基于 PIE 编译的, 所以被 linker 拦截下来了。

PIE 安全检查机制是 Google 的安全机制, 从 L 版本之后, 如果调用的可执行文件不是基于 PIE 方式编译的, 则无法运行。

编译时在 Android.mk 中加入如下 flag 即是基于 PIE 方式编译:

```
LOCAL_CFLAGS += -pie -fPIE
LOCAL_LDFLAGS += -pie -fPIE
```

根本原因:

libxguardian.so 应用父进程创建子进程, 但父进程先退出, 这样进程所在的进程组会成为孤儿进程组。

libuuid.so(豌豆荚)应用不是基于 PIE 编译导致 NE

解决方法:

1. 更新豌豆荚版本, 最新版本并没有这个问题

2. libxguardian.so 应用需修改架构

3. 符号缺失引起链接失败

问题背景:

打入 ViLTE patch 后 ims 注册失败

毕现

分析过程:

从 main log 看, 开机之后 VT Thread 一直都没有起来,

因为打开 ViLTE 后, IMS 一直在等待 VTSservice 来连接, 但是一直没有等到, 原因就是 VTSservice 进程一直起不来

```
Line 1649: [ 28.391873] (7)[1:init]init: Starting service 'vtservice'...
Line 1694: [ 28.512512] (7)[1:init]init: Service 'vtservice' (pid 3079) exited with status 1
Line 1695: [ 28.512548] (7)[1:init]init: Service 'vtservice' (pid 3079) killing any children in process group
Line 2852: [ 33.076339] (2)[1:init]init: Starting service 'vtservice'...
Line 2876: [ 33.338117] (5)[1:init]init: Service 'vtservice' (pid 3643) exited with status 1
Line 2877: [ 33.338167] (5)[1:init]init: Service 'vtservice' (pid 3643) killing any children in process group
Line 4000: [ 38.091726] (3)[1:init]init: Starting service 'vtservice'...
Line 4026: [ 38.196002] (0)[1:init]init: Service 'vtservice' (pid 4142) exited with status 1
Line 4027: [ 38.196031] (0)[1:init]init: Service 'vtservice' (pid 4142) killing any children in process group
Line 5282: [ 43.201721] (0)[1:init]init: Starting service 'vtservice'...
```

检查了 vtservice 都没有问题, 那么为何 vtservice 为自己退出呢? 怀疑是 vtservice 自己的行为, 因此加 log 到 main 函数, 结果一句 log 都没印出来。

起初以为 ALOGI 打印 log 函数被 logd 屏蔽掉了, 发现这个版本还未有该功能, 因此看起来连 main 函数都没跑到。

后面的调试方法是在 exit 函数里添加打印调用栈功能。按《[FAQ15114]如何获取进程的 native 调用栈?》添加代码, 结果 M 版本已不适用, 无法编译成功。

为了快速定位问题, 直接在 exit 里判断是 vtservice 后就直接 abort, 通过抓 coredump 分析。

修改代码如下:

```
char *strstr(const char *s, const char *find);
```

```

void exit(int status)
{

/* add this block */
if (strstr(getprogname(), "vtservice") && status == 1)
{
abort();
}
/* add end */

```

复现抓取了 ne db, 用 GAT 解开 db, 并结合对应的 symbols 文件 (symbols 目录里的文件必须和 db 一致), 利用工具 E-Consulter 分析, 分析报告如下:

== 异常报告 v2.9(仅供参考) ==

报告解读: MediaTek On-Line> Quick Start> E-Consulter 之 NE/KE 分析报告解读> NE 分析报告

详细描述: 从错误的地址 (0x00000000) 读数据, 请结合崩溃线程调用栈检查相关代码

版本 : alps-mp-m0.mp9/eng build

发生时间: Fri Jan 1 00:00:00 CST 2016

命令行 : /system/bin/vtservice

进程标识符(pid): 4051, 父进程标识符(ppid): 1 (/init)

进程状态: 正在运行

优先级 : 120 (0~99: 实时进程, 100~139: 普通进程)

== 线程信息(共 1 个线程) ==

当前线程信息:

线程名: /system/bin/vtservice, 暂停, 线程标识符(tid): 4051 (主线程)

本地调用栈:

linker __dl_strstr(s=0, find=0xF728CA4D) + 30 <bionic/libc/upstream-openbsd/lib/libc/string/strstr.c:49>

linker __dl_exit(参数 1=1) + 24 <bionic/libc/stdlib/exit.c:57>

linker __linker_init_post_relocation() + 520 <bionic/linker/linker.cpp:3219>

linker __dl__linker_init(raw_args=0xFFB4D890) + 830 <bionic/linker/linker.cpp:3426>

linker _start() + 4 <bionic/linker/arch/arm/begin.S:33>

== 栈结束 ==

对应汇编指令:

行号 地址 指令 提示

bionic/libc/upstream-openbsd/lib/libc/string/strstr.c

41 : F727E876: MOV R5, R0 ; __dl_strstr() 参数 1 可能有问题

49 : F727E88E: MOV R6, R5

F727E890: LDRB R3, [R6], #0x1! ; 线程停止在这里

当时的寄存器值:

R0: 00000008, R1: F728CA58, R2: 63697672, R3: 00000000, R4: F728CA4D, R5: 00000000, R6: 00000000, R7: FFB4D62

8

R8: 00000076, R9: 00000008, R10: FFB4D6AC, R11: FFB4D64C, R12: 80808000, SP: FFB4D628, R14: F727E88D, PC: F72

7E891

== C 堆检查 ==

分配器: jemalloc, 最多允许使用: 4GB, 最多使用: 0B, 当前使用: 0B, 泄露阈值: 128MB, 调试等级: 0

正常

== 日志信息 ==

main log:

01-01 00:00:00.308 4051 4051 F libc : Fatal signal 11 (SIGSEGV), code 1, fault addr 0x0 in tid 4051 (vtservice)

分析 NE 还需以下文件, 请提供 out/target/product/\$proj/symbols 目录下相应的文件(必须同一次编译生成, 如果是 lib 则选择/system/lib 目录下的文件):

libudf.so

vtservice

问题出在 linker, 先查看对应的代码:

```
3216 ElfW(Ehdr)* elf_hdr = reinterpret_cast<ElfW(Ehdr)*>(si->base);
3217 if (elf_hdr->e_type != ET_DYN) {
3218     __libc_format_fd(2, "error: only position independent executables (PIE) are supported.\n");
3219     exit(EXIT_FAILURE);
3220 }
```

看起来 vtservice 不是 PIE 引起的 exit(1), 马上查看 vtservice, 通过 readelf 看是否是 DYN, 命令如下:

- `arm-linux-android-readelf -aW vtservice`

看到是 DYN 类型的, 那就是 PIE 了, 为何在这里报错, 怀疑和:

- [MediaTek On-Line](#)> [Quick Start](#)> 踩内存专题分析> native 案例分析> TEE 踩坏浮点寄存器引起 SF NE

相关, 结果确认了是 tbase TEE, 没有这个问题。

那是怎么回事? 看到 linker 的 log 是直接输出到 stderr 的, 那么直接在 adb shell 输入:

- `#!/system/bin/vtservice`

就可以看到是什么错误信息了, 结果看到:

```
CANNOT LINK EXECUTABLE: cannot locate symbol "_ZN7android25MakeHEVCCodecSpecificDataEPKcPiS2_" referenced by "/system/lib/libimsma.so"...
```

原来是符号找不到引起的 link 失败。为何之前会导向不是 PIE 错误呢，原因是编译器优化了，将所有 exit(1) 合并在一起，无从知道是哪里跳过来的了。

通过内部查找，发现这个符号是在 libcomutils.so，检查出问题的版本的 libcomutils.so，用 readelf 查看符号：

- arm-linux-android-readelf -aW libcomutils.so

搜索 _ZN7android25MakeHEVCCodecSpecificDataEPKcPiS2_，结果确实没找到。

经过确认发现某个 patch 开始就存在这个符号了，不可能缺失。下载对应的 patch，用上面的方法查看，发现有这个符号的。

那么问题就明显了，是没有合入这个 so 导致的问题。合入这个 patch 后发现还是有其他符号找不到，需要彻查所有 patch 合入的情况。

根本原因：

patch 没有全部合入引起 vtsservice 链接失败。

解决方法：

拿到的 patch 务必全部合入。

结语：

需要对 linker、elf 熟悉。

4. 环境变量引起 linker 时寻找库错误“is 32-bit instead of 64-bit”

问题背景：

经常出现下面的 NE

```
Revision: '0'
ABI: 'arm64'
pid: 9544, tid: 9544, name: ls >>> ls <<<
signal 6 (SIGABRT), code -6 (SI_TKILL), fault addr -----
x0 0000000000000000 x1 0000000000002548 x2 0000000000000006 x3 0000000000000008
x4 ffffffffefefefef x5 0000000000000000 x6 0000008080808080 x7 2c33351f656e1f63
x8 0000000000000083 x9 ffffffffefefefdf x10 0000000000000000 x11 0000000000000001
x12 0000000000000018 x13 0000000000000000 x14 0000000000000000 x15 0034645024a4b2f3
x16 0000000000000003 x17 2207397d3179729a x18 000000000000003a x19 0000007b96195b40
x20 0000000000000006 x21 0000007b96195a98 x22 0000000000000000 x23 0000000000000005
x24 0000000000000001 x25 0000007fd208ce10 x26 0000007fd208ce40 x27 0000007b960ce010
x28 0000007b960d7811 x29 0000007fd208cc80 x30 0000007b96156f60
```

```
12-24 00:00:03.878 9544 9544 F libc : CANNOT LINK EXECUTABLE "ls": "/system/vendor/lib/libdirect-coreddump.so" is  
32-bit instead of 64-bit
```

```
12-24 00:00:03.887 9544 9544 F libc : Fatal signal 6 (SIGABRT), code -6 in tid 9544 (ls)
```

```
12-24 00:00:03.891 1773 1773 D AEE_AED : $===AEE===AEE===AEE===  
12-24 00:00:03.891 1773 1773 D AEE_AED : p 2 poll events 1 revents 1
```

分析过程：

```
12-24 00:00:03.878 9544 9544 F libc : CANNOT LINK EXECUTABLE "ls": "/system/vendor/lib/libdirect-coreDump.so" is  
32-bit instead of 64-bit
```

```
12-24 00:00:03.887 9544 9544 F libc : Fatal signal 6 (SIGABRT), code -6 in tid 9544 (ls)
```

```
12-24 00:00:03.891 1773 1773 D AEE_AED : $===AEE===AEE===AEE===
```

让客户从手机中看到底有没有这个库，看能不能执行 `ls` 命令，客户的回答是肯定可以的。那就只好看 linker 的源码了，为什么会找错库。

先看看出错的位置

```
bool ElfReader::VerifyElfHeader() {
    ....

    if (needed_libraries_count > 0 &&
        !find_libraries(&g_default_namespace, si, needed_library_names, needed_libraries_count,
            nullptr, &g_ld_preloads, ld_preloads_count, RTLD_GLOBAL, nullptr,
            /* add_as_children */ true)) {
        __libc_fatal("CANNOT LINK EXECUTABLE \"%s\": %s", args.argv[0], linker_get_error_buffer());
    } else if (needed_libraries_count == 0) {
        .....
    }
}
```

再看一下函数的调用流程:

```
__linker_init_post_relocation()->find_libraries()->find_library_internal()->load_library()  
->open_library()
```

然后在 open_library() 中看到下面一段代码.

```
// Otherwise we try LD_LIBRARY_PATH first, and fall back to the default library path  
int fd = open_library_on_paths(zip_archive_cache, name, file_offset, ns->get_ld_library_paths(), realpath);  
if (fd == -1 && needed_by != nullptr) {  
    fd = open_library_on_paths(zip_archive_cache, name, file_offset, needed_by->get_dt_runpath(), realpath);  
    // Check if the library is accessible  
    if (fd != -1 && !ns->is_accessible(*realpath)) {  
        fd = -1;  
    }  
}
```

注释已经写得很清楚了, 先去尝试 LD_LIBRARY_PATH 这个路径, 再去试 default library path.

我们怎么知道 LD_LIBRARY_PATH 这个值是多少了?

在 db 下面 PROCESS_ENVIRONMENT 这个文件中有记录, 进程出现 NE 时, PROCESS_ENVIRONMENT 这个文件会把当时的环境变量记录下来。

我们在 PROCESS_ENVIRONMENT 中可以看到

```
LD_LIBRARY_PATH=/vendor/lib:/system/lib
```

所以找到 32 位的库而没有找 64 位的。

那为什么 LD_LIBRARY_PATH 这个值会是 /vendor/lib:/system/lib? 那这个就是当前进程可以设置的。

我们再看看当时出现问题的进程。

u:r:untrusted_app:s0:c512,c768 u0_a192	9544	9537	1852	756	1	30	10	0	0	1
s										
u:r:untrusted_app:s0:c512,c768 u0_a192	9537	32496	8088	2496	1	30	10	0	0	sh
u:r:untrusted_app:s0:c512,c768 u0_a192	32496	645	1203800	99704	2	20	0	0	0	co
m.tencent.qqvim										

此因可以断定是 com.tencent.qqvim 这个有问题。为了验证这个, 就改了一个应用程序.

```
while(1) {  
    sleep(10);  
    printf("system ls start\n");  
    putenv("LD_LIBRARY_PATH=/vendor/lib:/system/lib");  
}
```

```
printf("%s 2\n",getenv("LD_LIBRARY_PATH"));  
system("/system/bin/sh -c ls");  
sleep(50);  
printf("system ls end\n");  
}
```

运行一下，出现相同的 NE db。

根本原因：

应用程序本身的原因,可能版本太旧。

解决方法：

拿掉旧的应用程序

结语：

需要对 linker 熟悉。

5. 库包库导致 app 崩溃

问题背景

在安装使用安徽移动 APP 过程中发现，该 APP 一连网，就会 Force Close，用 eng 版本复现，抓到 NE db，里面包含 coredump。

分析过程

利用工具 E-Consulter 分析，产生分析报告如下：

== 异常报告 v3.2 (仅供参考) ==

报告解读: MediaTek On-Line> Quick Start> E-Consulter 之 NE/KE 分析报告解读> NE 分析报告

详细描述: 从错误的地址 (0x00000000) 读数据，请结合崩溃线程调用栈检查相关代码

版本 : alps-mp-n0.mpl/eng build

发生时间: Mon Dec 18 11:02:45 CST 2017

命令行 : com.sitech.ac

pid : 10334, ppid: 345 (zygote)

进程状态: 正在运行

优先级 : 120 (0~99: 实时进程, 100~139: 普通进程)

== 线程信息 (共 82 个线程) ==

当前线程信息:

线程名: main, 可中断睡眠, tid: 10334 (主线程)

errno: 2

线程状态: 在调用 JNI 方法

本地调用栈:

libutils.so android::SharedBuffer::acquire() <external/libcxx/include/atomic:930>

libutils.so android::allocFromUTF8() + 76 <system/core/libutils/String16.cpp:37>

```

libutils.so android::String16::String16(参数 1=0xBECB557C) + 8 <system/core/libutils/String16.cpp:160>
libdatabase_sqlcipher.so 0x90FBC51D(参数 1=0xAB8993F0, 参数 2=0xBECB55DC, 参数 3=0, 参数 4=6) + 98
base.odex net.sqlcipher.CursorWindow.getString_native() + 92
/dev/ashmem/dalvik-jit-code-cache 0x95EE4433()
/dev/ashmem/dalvik-jit-code-cache 0x95EE4F4F(参数 2=0x12E833C0, 参数 3=6)
system@framework@boot-framework.oat android.database.CursorWrapper.getString(参数 1=0x6FE03554) + 46
libart.so art_quick_invoke_stub_internal() + 64
.....
对应汇编指令:
行号 地址 指令 提示
external/libcxx/include/atomic
930 : 92275462: LDXR R1, [R0] ; android::SharedBuffer::acquire() 参数 1 可能有问题
当时的寄存器值:
R0: 00000000, R1: 00000000, R2: 00000000, R3: 8F700000, R4: 00000000, R5: 8F700137, R6: AC0D5008, R7: 904B8AB
8
R8: 12E21678, R9: AB884400, R10: 00000000, R11: BECB571C, R12: 92282C4C, SP: BECB5550, R14: 9227681D, PC: 922
75463

```

可以看到 android::SharedBuffer::acquire() 参数 1 = NULL 导致了 NE。参数 1 来自 system/core/libutils/String16.cpp:

```

static inline char16_t *getEmptyString()
{
    gEmptyStringBuf->acquire(); /* 这里调用了 acquire(), 其中 gEmptyStringBuf = NULL 导致了 NE */
    return gEmptyString;
}

```

我们发现 gEmptyStringBuf 是全局静态变量，是由 initialize_string16() 初始化的:

```

void initialize_string16()
{
    SharedBuffer* buf = SharedBuffer::alloc(sizeof(char16_t));
    char16_t* str = (char16_t*)buf->data();
    *str = 0;
    gEmptyStringBuf = buf;
    gEmptyString = str;
}

```

而 initialize_string16() 是被如下函数调用:

```

class LibUtilsFirstStatics
{
public:
    LibUtilsFirstStatics()
    {

```



```

        initialize_string8();
        initialize_string16();
    }
    ~LibUtilsFirstStatics()
    {
        terminate_string16();
        terminate_string8();
    }
};

static LibUtilsFirstStatics gFirstStatics;

```

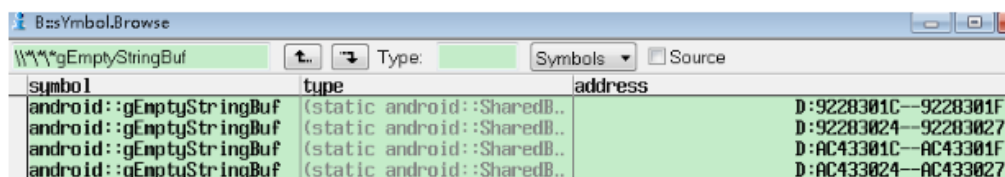
在 LibUtilsFirstStatics 的构造函数里调用了 initialize_string16()。由于定义了 gFirstStatics 全局静态对象，这个对象只有一个用处，那就是在库加载时调用构造函数而已。因此 gEmptyStringBuf 在 libutils.so 加载时初始化。

可是为何在这个进程里的 gEmptyStringBuf = NULL 呢？是否可能是：

- 库加载时没有调用 LibUtilsFirstStatics() 或 initialize_string16()
- 库卸载时调用了 ~LibUtilsFirstStatics()，释放了 gEmptyStringBuf。
- gEmptyStringBuf 被踩坏

我们在分析报告里搜索了 link 关键字，没有找到任何对应的函数，因此排除第 2 点。

而第 1 点，更不可能了，难道 linker 坏掉了？从第 3 点出发，用 trace32 加载 debug.cmm，看下 gEmptyStringBuf：



symbol	type	address
android::gEmptyStringBuf	(static android::SharedB..	D:9228301C--9228301F
android::gEmptyStringBuf	(static android::SharedB..	D:92283024--92283027
android::gEmptyStringBuf	(static android::SharedB..	D:AC43301C--AC43301F
android::gEmptyStringBuf	(static android::SharedB..	D:AC433024--AC433027

居然有 4 个 gEmptyStringBuf，不合理，后来发现 1 个 libutils.so 就有 2 个 gEmptyStringBuf，1 个是 String8 的，1 个是 String16 的。那为何有 4 个？查看 PROCESS_MAPS：

```

Line 896: 92268000-92281000 r-xp 00000000 b3:15 1146 /system/lib/libutils.so
Line 898: 92282000-92283000 r--p 00019000 b3:15 1146 /system/lib/libutils.so
Line 899: 92283000-92284000 rw-p 0001a000 b3:15 1146 /system/lib/libutils.so
Line 1475: ac418000-ac431000 r-xp 00000000 b3:15 1146 /system/lib/libutils.so
Line 1477: ac432000-ac433000 r--p 00019000 b3:15 1146 /system/lib/libutils.so
Line 1478: ac433000-ac434000 rw-p 0001a000 b3:15 1146 /system/lib/libutils.so

```

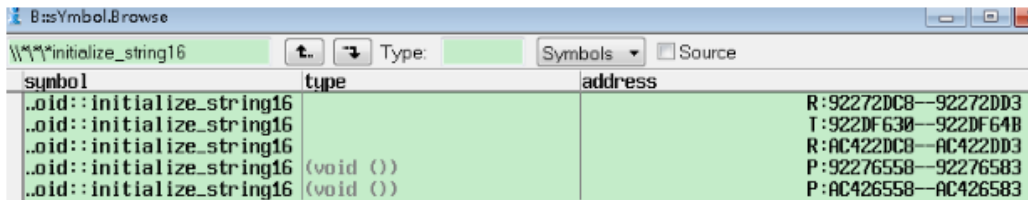
发现 libutils.so 被加载了 2 次，这是为何？这是 Android N 及之后 linker 的一个新功能：namespace。namespace 用于控制 native lib 的链接，避免 app 随意使用私有 native API。如果用不好就会导致不同的 namespace 可能加载同样的库。

好了，回到正题，即使加载了 2 次，4 个 gEmptyStringBuf 也应该被初始化，但打开变量发现都没被初始化，这就诡异了。

难道是 linker 出问题了（怀疑到第 1 点）？其实这是 linker 的链接规则引起的问题，有些库利用这个规则 hook。

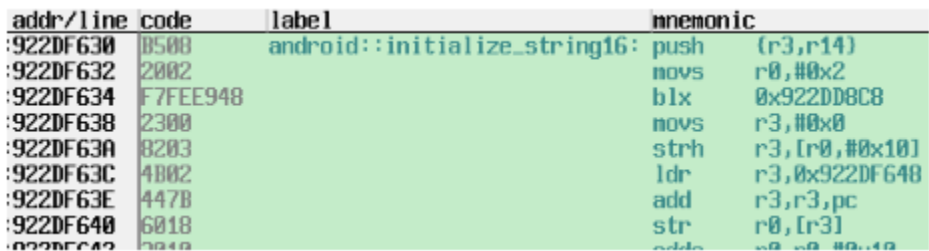
比如 libudf.so，里面同样定义了和 libc.so 的 malloc/free 等函数，程序在调用 malloc 函数时，不是调用了 libc.so 的 malloc，而是转调用到 libudf.so 的 malloc，达到 hook 目的，用于 debug malloc。

于是我们找下是否有人定义同样的函数 `initialize_string16()`:



symbol	type	address
..oid::initialize_string16		R:92272DC8—92272DD3
..oid::initialize_string16		T:922DF630—922DF64B
..oid::initialize_string16		R:AC422DC8—AC422DD3
..oid::initialize_string16	(void ())	P:92276558—92276583
..oid::initialize_string16	(void ())	P:AC426558—AC426583

我们发现除了 2 个 `libutils.so` 之外还有一处定义了 `initialize_string16()`:



addr/line	code	label	mnemonic
:922DF630	0500	android::initialize_string16:	push {r3,r14}
:922DF632	2002		movs r0,#0x2
:922DF634	F7FEE948		blx 0x922DD8C8
:922DF638	2300		movs r3,#0x0
:922DF63A	8203		strh r3,[r0,#0x10]
:922DF63C	4B02		ldr r3,0x922DF648
:922DF63E	447B		add r3,r3,pc
:922DF640	6018		str r0,[r3]
:922DF642	0010		add r0,r0,#0x10

查看 `0x922DF630` 是属于哪个 lib:

```
92284000-92498000 r-xp 00000000 b3:17 1703955 /data/app/com.sitech.ac-1/lib/arm/libsqlcipher_android.so
92498000-924ab000 r--p 00213000 b3:17 1703955 /data/app/com.sitech.ac-1/lib/arm/libsqlcipher_android.so
924ab000-924ad000 rw-p 00226000 b3:17 1703955 /data/app/com.sitech.ac-1/lib/arm/libsqlcipher_android.so
```

终于找到凶手了，是 `libsqlcipher_android.so`。 `libsqlcipher_android.so` 为何定义 `initialize_string16()`?

怀疑 `libsqlcipher_android.so` 静态连接了 `libutils.so`，部分 symbol 被 export 出来，扰乱了链接顺序。

原本应该是:

1. `0x92268000` 地址处的 `libutils.so` 加载时调用自身的 `initialize_string16()` 函数初始化自身的 `gEmptyStringBuf`
2. `0xac418000` 地址处的 `libutils.so` 加载时调用自身的 `initialize_string16()` 函数初始化自身的 `gEmptyStringBuf`

但由于 `libsqlcipher_android.so` 也定义了 `initialize_string16()`，导致变成了:

1. `0x92268000` 地址处的 `libutils.so` 加载时调用 `libsqlcipher_android.so` 的 `initialize_string16()` 函数初始化 `libsqlcipher_android.so` 的 `gEmptyStringBuf`
2. `0xac418000` 地址处的 `libutils.so` 加载时调用 `libsqlcipher_android.so` 的 `initialize_string16()` 函数初始化 `libsqlcipher_android.so` 的 `gEmptyStringBuf`
3. `0x92268000` 地址处的 `libutils.so` 里的 `gEmptyStringBuf` 和 `0xac418000` 地址处的 `libutils.so` 里的 `gEmptyStringBuf` 都没有初始化

额外的问题是不应该加载 2 次 `libutils.so`，应该是 app 本身使用私有化 native API。

根本原因

有 2 个原因:

- 应用本身携带的库有问题，扰乱链接。

- libutils.so 被 Mediatek 修改过, 导致 initialize_string16() 被 export 出去, 如果没有 export 出去就不会有问题了。patch: ALPS02995763, 将还原为 AOSP 的做法。如下是 AOSP 的 libutils.so (左边) 和有问题的 libutils.so (右边) 的 initialize_string16() 符号的属性, DEFAULT 将被 export 出去:

GLOBAL PROTECTED	13	_ZN7android6LooperD2Ev	GLOBAL DEFAULT	13	_ZN7android6LooperD2Ev
GLOBAL PROTECTED	13	_ZN7android19initialize_string16Ev	GLOBAL DEFAULT	13	_ZN7android19initialize_string16Ev
GLOBAL PROTECTED	13	_ZN7android7String8C1ERK30_	GLOBAL DEFAULT	13	_ZN7android7String8C1ERK30_

通过 .rel.plt 可以明确看到有问题的 libutils.so 的 initialize_string8 等需要链接, 而 AOSP 的 libutils.so 则不需要!

```
Relocation section '.rel.plt' at offset 0x9e44 contains 341 entries:
Offset      Info      Type           Sym. Value  Symbol's Name
.....
0001ac38 0001ac16 R_ARM_JUMP_SLOT 0000e585   _ZN7android18terminate_string16Ev
0001ac3c 0001ce16 R_ARM_JUMP_SLOT 0000d76d   _ZN7android17terminate_string8Ev
0001ac40 00018c16 R_ARM_JUMP_SLOT 0000d729   _ZN7android18initialize_string8Ev
0001ac44 0001dc16 R_ARM_JUMP_SLOT 0000e559   _ZN7android19initialize_string16Ev
```

解决方法

请第 3 方修复或者 patch ALPS02995763

结语

需要熟悉 linker, 熟悉 namespace。

6. ART NE

问题背景:

执行开关机测试, 发生多笔 system server NE

分析过程:

通过工具解析的分析报告如下:

报告解读: MediaTek On-Line> Quick Start> E-Consulter 之 NE/KE 分析报告解读> NE 分析报告

详细描述: 从错误的地址 (0xFFFFFFFFFFFFF8) 读数据, 请结合崩溃线程调用栈检查相关代码

版本 : alps-mp-o1.mp7/user build

发生时间: Sun Jun 24 07:31:16 CST 2018

命令行 : system_server

pid : 838, ppid: 500 (zygote64)

进程状态: 不可中断睡眠

优先级 : 118 (0~99: 实时进程, 100~139: 普通进程)

本地调用栈:

```
libart.so art::DumpKernelStack(参数 1=0x0000007612CCC2B0, 参数 2=2430, 参数 3=0x000000761B550F2F, 参数 4=0) + 160
<external/libcxx/include/string:1226>
libart.so art::DumpUnattachedThread(os=0x0000007612CCC2B0) + 340 <art/runtime/thread_list.cc:164>
libart.so art::ThreadList::DumpUnattachedThreads(this=0x000000761BB09000, os=0x0000007612CCC2B0,
dump_native_stack=false) + 612 <art/runtime/thread_list.cc:189>
libart.so art::ThreadList::DumpForSigQuit(this=0x000000761BB09000, os=0x0000007612CCC2B0) + 928
<art/runtime/thread_list.cc:156>
```

```

libart.so art::Runtime::DumpForSigQuit(this=0x000000761BABD600, os=0x0000007612CCC2B0) + 196
<art/runtime/runtime.cc:1677>
libart.so art::SignalCatcher::HandleSigQuit(this=0x0000007612839960) + 1936 <art/runtime/signal_catcher.cc:194>
libart.so art::SignalCatcher::Run(arg=0x0000007612839960) + 348 <art/runtime/signal_catcher.cc:271>
libc.so 0x000000769C88C0E4(参数 1=0x0000007612CCC4F0) + 36
libc.so 0x000000769C842DB8() + 68
== 栈结束 ==

```

NE 在 art/runtime/native_stack_dump.cc:388

kernel_stack_frames.pop_back(); //访问 0xFFFFFFFFFFFFFFF8 这个错误的地址

查看 kernel_stack_filename 结构体:

```

kernel_stack_filename = (
    __r_ = (
        std::__1::__libcpp_compressed_pair_imp<std::__1::basic_string<char, std::__1::char_traits<char>,
std::__1::allocator<ch
        __l = (__cap_ = 0x31, __size_ = 0x1A, __data_ = 0x00000078D8BC9D30 = end+0x1376D10 ->
"/proc/self/task/2718/stack"),
        __s = (__size_ = 0x31, __lx = 0x31, __data_ = ""),
        __r = (__words = (0x31, 0x1A, 0x00000078D8BC9D30))))))
kernel_stack = (
    __r_ = (
        std::__1::__libcpp_compressed_pair_imp<std::__1::basic_string<char, std::__1::char_tra
        __l = (__cap_ = 0x0, __size_ = 0x0, __data_ = 0x0 = -> NULL),
        __s = (__size_ = 0x0, __lx = 0x0, __data_ = ""),
        __r = (__words = (0x0, 0x0, 0x0))))))
kernel_stack_frames = (
    std::__1::__vector_base<std::__1::basic_string<char, std::__1::char_traits<char>,
std::__1::allocator<char> >, std::__1::
    std::__1::__vector_base<std::__1::basic_string<char, std::__1::char_traits<char>,
std::__1::allocator<char> >, std::__1::allocator<std::__1::basic_string<char, std::__1::char_traits<char>,
std::__1::allocator<char> > >::__end_ = 0xFFFFFFFFFFFFFFF8

```

当前 dump 的 thread 是 2718 这个线程，而 tid: 2718 在 SYS_PROCESSES_AND_THREADS 这个文件中已经找不到了

从 bsp log 看到在 NE 前 1-2s，线程 2718 退出了：

```
-2718 [002] .... 31.652599: sched_process_exit: comm=SharedPreferenc pid=2718 prio=120
```

所以 ReadFileToString(kernel_stack_filename, &kernel_stack) 执行完 kernel_stack 应该都是空的。

从以上分析推测，可能是线程当时恰好退出然后抓该线程的 stack，导致 NE。为了验证此问题确实是因为 2718 线程退出导致，请客户将/art/runtime/native_stack_dump.cc 的以下 379, 380, 381, 382 这四行注释掉，然后发 signal 3 给 system server 进程，结果触发同样的 NE

```
371 void DumpKernelStack(std::ostream& os, pid_t tid, const char* prefix, bool include_count) {
372   if (tid == GetTid()) {
373     // There's no point showing that we're reading our stack out of /proc!
374     return;
375   }
376
377   std::string kernel_stack_filename(StringPrintf("/proc/self/task/%d/stack", tid));
378   std::string kernel_stack;
379   if (!ReadFileToString(kernel_stack_filename, &kernel_stack)) {
380     os << prefix << "(couldn't read " << kernel_stack_filename << ")\n";
381     return;
382   }
```

根本原因：

检查 ReadFileToString() 这个 function，下面红色这里返回 true 的本意是如果最后 read() return 0 的时候，就表示资料读完了，例如，read() 会依次 return 8K, 8K, 4K, 0，最后 read() return 0 的时候，这个 function 就 return 了，但是没有考虑到 read() 第一次就 return 0 的情况。

```
79 File file(file_name, O_RDONLY, false);
80 if (!file.IsOpened()) {
81   return false;
82 }
83
84 std::vector<char> buf(8 * KB);
85 while (true) {
86   int64_t n = TEMP_FAILURE_RETRY(read(file.Fd(), &buf[0], buf.size()));
87   if (n == -1) {
88     return false;
89   }
90   if (n == 0) {
91     return true;
92   }
93   result->append(&buf[0], n);
94 }
```

从 code 来看，本来应该是假设 File file(file_name, O_RDONLY, false); 有成功，就应该可以读到资料，但就在 open -> read 的时间，thread 就 exit 了

解决方法：

```
if (n == 0) {
```

```

if(0==result->size())

    return false;

return true;

}

```

7. Pthread Key 填满问题分析

问题背景:

一家客户反馈，常时间跑 camera 的压力测试，cameraserver 可能 crash.

分析过程:

1. 与客户确认问题状态，客户一开始只提交了只抓到了一个 main log，可以看到 cameraserver 因为 NE 退出。为了确认问题点，请客户复现抓到了 cameraserver 的 coredump。并且提供了一个不匹配的第三方 symbols，以及其他的 symbols.
2. 从 coredump 解析.coredump，最后 crash 是因为:

```

libc.so                                abort(参数 4=key_map + 1120) + 4 <bionic/libc/arch-arm/bionic/abort_ar
m.S:43>
libst_personblur.so                    emutls_init() + 16
libc.so                                pthread_once() + 40 <bionic/libc/bionic/pthread_once.cpp:71>
libst_personblur.so                    __emutls_get_address(obj=0xCCFC7780) +68

```

根据客户的 symbols 利用函数匹配可以抓到对应的汇编:

```

symbol libst_personblur.so, 实际是因为:
ZSR:0067D038|E92D4008  emutls_init:          push    {r3,r14}

      |
      |
      | 87
      |
ZSR:0067D03C|EBFFFFFF          b1      0x67D018      ; __thread_key_create.constprop.2
ZSR:0067D040|E3500000          cmp     r0,#0x0      ; r0,#0
ZSR:0067D044|088D0000          popeq   {r3,pc}
      |
      | 88
      |
ZSR:0067D048|EBE6A508          b1      0x267BC

      |
      |
      | 709
      |
ZSR:0067D04C|E92D4008  __thread_once.constprop.3:  push    {r3,r14}

      |
      | 711
      |
ZSR:0067D050|EBFFFFFF          b1      0x67CEFB      ; __thread_active_p
ZSR:0067D054|E3500000          cmp     r0,#0x0      ; r0,#0
ZSR:0067D058|0A000000          beq     0x67D078

```

即 call __thread_key_create => pthread_key_create 后，check 返回值错误，然后主动 call abort 重启.

而 pthread_key_create 失败的原因是，因为 pthread key 已经塞满了。解析 pthread 的 key_map 可以看到完整的 pthread key 的情况，特别是知道它的析构函数.

```

(seq = 0x1, key_destructor = 0xCCC58350),
(seq = 0x1, key_destructor = 0xCCC58350),
(seq = 0x1, key_destructor = 0xCCC58350),
(seq = 0x1, key_destructor = 0xCCC58350),
(seq = 0x1, key_destructor = 0xCCC58350),

```


[illegible]

[illegible]

```
(seq = 0x1, key_destructor = 0xCCC58350),
(seq = 0x1, key_destructor = 0xCCC58350),
(seq = 0x1, key_destructor = 0xCCC58350),
(seq = 0x1, key_destructor = 0xCCC58350),
(seq = 0x1, key_destructor = 0xCCC58350),
(seq = 0x1, key_destructor = 0xCCC58350),
(seq = 0x1, key_destructor = 0xCCC58350),
(seq = 0x1, key_destructor = 0xCCC58350),
(seq = 0x1, key_destructor = 0xCCC58350),
(seq = 0x1, key_destructor = 0xCCC58350),
(seq = 0x1, key_destructor = 0xCCC58350),
(seq = 0x1, key_destructor = 0xCCC58350),
(seq = 0x1, key_destructor = 0xCCC58350),
(seq = 0x1, key_destructor = 0xCCC58350),
(seq = 0x1, key_destructor = 0xCCC58350),
(seq = 0x1, key_destructor = 0xCCC58350),
.....
```

可以看到大面积的 pthread key 都是同样的一个析构函数 0xCCC58350, 然后我们解析这个析构函数的汇编:

```
cc5df000-ccdb4000 r-xp 00000000 103:02
4854 /system/vendor/lib/libst_personblur.so
ccdb4000-ccdc7000 r--p 007d4000 103:02
4854 /system/vendor/lib/libst_personblur.so
ccdc7000-ccfcf000 rw-p 007e7000 103:02
4854 /system/vendor/lib/libst_personblur.so
```

可以看到:

	NUD:CCC58350 E92D40F8	push	{r3-r7, r14}	
	NUD:CCC58354 E5907000	ldr	r7, [r0]	
	NUD:CCC58358 E1A04000	cpy	r4, r0	
	NUD:CCC5835C E1A06000	cpy	r6, r0	
	NUD:CCC58360 E3A05000	mov	r5, #0x0	; r5, #
0				
	NUD:CCC58364 E1550007	cmp	r5, r7	
	NUD:CCC58368 0A000006	beq	0xCCC58388	
	NUD:CCC5836C E5B63004	ldr	r3, [r6, #0x4]!	
	NUD:CCC58370 E3530000	cmp	r3, #0x0	; r3, #
0				
	NUD:CCC58374 0A000001	beq	0xCCC58380	
	NUD:CCC58378 E5130004	ldr	r0, [r3, #0xFFFFFFFFC]	
	NUD:CCC5837C EBE6B4D1	bl	0xCC6056C8	; free
	NUD:CCC58380 E2855001	add	r5, r5, #0x1	; r5, r5, #
1				
	NUD:CCC58384 EAF7FFF6	b	0xCCC58364	

	NUD:CCC58388	E1A00004	cpy	r0, r4	
	NUD:CCC5838C	E8BD40F8	pop	{r3-r7, r14}	
	NUD:CCC58390	EAE6B4CC	b	0xCC6056C8	; free
	NUD:CCC58394	EAE6B645	b	0xCC605CB0	; pthre
ad_getspecific					
	NUD:CCC58398	EAE6B5CF	b	0xCC605ADC	; pthre
ad_setspecific					
	NUD:CCC5839C	E92D4070	push	{r4-r6, r14}	
	NUD:CCC583A0	E5905004	ldr	r5, [r0, #0x4]	
	NUD:CCC583A4	E1A06000	cpy	r6, r0	
	NUD:CCC583A8	E3550004	cmp	r5, #0x4	; r5, #
4					
	NUD:CCC583AC	E5900000	ldr	r0, [r0]	
	NUD:CCC583B0	8A000006	bhi	0xCCC583D0	
	NUD:CCC583B4	E2800004	add	r0, r0, #0x4	; r0, r0, #
4					
	NUD:CCC583B8	EBE6B4BC	bl	0xCC6056B0	; malloc
	NUD:CCC583BC	E3500000	cmp	r0, #0x0	; r0, #
0					
	NUD:CCC583C0	15800000	strne	r0, [r0]	
	NUD:CCC583C4	12804004	addne	r4, r0, #0x4	; r4, r0, #4
	NUD:CCC583C8	1A00000A	bne	0xCCC583F8	

再次利用函数匹配的方式:

	ZSR:0067CF00	E92D40F8	emutls_destroy:		pus
h		{r3-r7, r14}			
		69			
	ZSR:0067CF04	E590700			
0			ldr	r7, [r0]	
		67			
	ZSR:0067CF08	E1A0400			
0			cpy	r4, r	
0					
		; r4			
	ZSR:0067CF0C	E1A0600			
0			cpy	r6, r	
0					
		; r6			

		72		
	ZSR:0067CF10 E3A0500			
0		mov	r5, #0x	
0	; r5			
	ZSR:0067CF14 E155000			
7		cmp	r5, r	
7	; i,			
	ZSR:0067CF18 0A00000			
6		beq	0x67CF38	
		74		
	ZSR:0067CF1C E5B6300			
4		ldr	r3, [r6, #0x4]!	
	ZSR:0067CF20 E353000			
0		cmp	r3, #0x	
0	; r3			
	ZSR:0067CF24 0A00000			
1		beq	0x67CF30	

Root cause:

即这个问题的原因应当是，vendor 在线程 HPTD_DQ_DIP1 启动的时候，通过 emutls_init call 了 __gthread_key_create ==> pthread_key_create，但在线程退出的时候忘记删除 key 了，这个导致 pthread key 累计越来越多，最后爆掉了，而对应 key 的析构函数是 emutls_destroy。麻烦 vendor 修正。

8. 错误 munmap 别人的内存引起 NE

问题背景

MTBF 测试随机打中各种 ART NE 问题。

分析过程

分析几个 db，利用工具 SpOfflineDebugSuite 分析，产生分析报告如下：

== 异常报告 v2.0(仅供参考) ==

报告解读: MediaTek On-Line> Quick Start> NE/KE 分析报告解读> NE 分析报告

详细描述: 从错误的地址(0x0000007632AA3000)读数据，请结合崩溃线程调用栈检查相关代码

版本 : alps-mp-p0.mpl/userdebug build

发生时间: Fri Oct 19 09:30:00 CST 2018

命令行 : com.android.systemui

pid : 8857, ppid: 12798 (zygote64)

== 线程信息(共 66 个线程) ==

当前线程信息:

线程名: Binder:8857_B, 可中断睡眠, tid: 27728

errno: 22

线程状态: 正在运行

Java 调用栈:

..... java.lang.ref.Reference.getReferent() <本地方法>

boot.oat java.lang.ref.Reference.get()

boot.oat java.lang.ThreadLocal\$ThreadLocalMap.set()

boot.oat java.lang.ThreadLocal.set()

boot-framework.oat android.os.Binder.execTransact()

== 栈结束 ==

本地调用栈:

libart.so art::IrtEntry::Add() + 12 <art/runtime/indirect_reference_table-inl.h:110>

libart.so art::IndirectReferenceTable::Add(this=0x000000762C01FE00, error_msg=0x000000760F1E6950) + 1204 <art/runtime/indirect_reference_table.cc:313>

libart.so art::JNIEnvExt::AddLocalReference(<_jobject*>)(this=0x000000762C01FDE0, obj=0x000000760F1E69A4) + 60 <art/runtime/jni_env_ext-inl.h:29>

system@framework@boot.oat java.lang.ref.Reference.getReferent() + 124 <本地方法>

system@framework@boot.oat java.lang.ref.Reference.get(参数 2=0x00000000131B74D8) + 40

system@framework@boot.oat java.lang.ThreadLocal\$ThreadLocalMap.set(参数 2=0x00000000131B7470, 参数 3=0x0000000075A1EC20, 参数 4=0) + 140

system@framework@boot.oat java.lang.ThreadLocal.set(参数 2=0x0000000075A1EC20, 参数 3=0) + 96

system@framework@boot-framework.oat android.os.Binder.execTransact() + 1260

libart.so art_quick_invoke_stub(参数 1=0x00000000717267D0, 参数 2=0x000000760F1E6E60, 参数 3=28, 参数 4=0x000000762C088800, 参数 5=0x000000760F1E6E40, 参数 6=0x0000000074E0CC07) + 584 <art/runtime/arch/arm64/quick_entrypoints_arm64.S:1702>

libart.so art::ArtMethod::Invoke(this=0x00000000717267D0, self=0x000000762C088800, args=0x000000760F1E6E60, args_size=28, result=0x000000760F1E6E40, shorty=0x0000000074E0CC07) + 200 <art/runtime/art_method.cc:374>

libart.so art::(anonymous namespace)::InvokeWithArgArray(参数 1=0x000000760F1E6F48, 参数 2=0x00000000717267D0, 参数 3=0x000000760F1E6E48, 参数 4=0x000000760F1E6E40, 参数 5=0x0000000074E0CC07) + 104 <art/runtime/reflection.cc:456>

libart.so art::InvokeVirtualOrInterfaceWithVarArgs(参数 1=0x000000760F1E6F48, 参数 2=10118, 参数 3=0x00000000717267D0, 参数 4=0x000000760F1E6F10) + 432 <art/runtime/reflection.cc:580>

libart.so art::JNI::CallBooleanMethodV(obj=10118, mid=0x00000000717267D0, args=0x000000760F1E70A0) + 648 <art/runtime/jni_internal.cc:844>

libandroid_runtime.so _JNIEnv::CallBooleanMethod(this=0x000000762C01FDE0, obj=10118) + 120 <libnativehelper/include_jni/jni.h:620>

libandroid_runtime.so JavaBBinder::onTransact(this=0x0000007628A6BC40, code=1, data=0x000000760F1E7228, reply=0x000000760F1E71C0, flags=17) + 156 <frameworks/base/core/jni/android_util_Binder.cpp:344>

libbinder.so android::BBinder::transact(this=0x0000007628A6BC40, code=1, data=0x000000760F1E7228, reply=0x000000760F1E71C0, flags=17) + 136 <frameworks/native/libs/binder/Binder.cpp:129>

libbinder.so android::IPThreadState::executeCommand(this=0x000000762C03A8C0, cmd=-2143260158) + 520 <frameworks/native/libs/binder/IPThreadState.cpp:1121>

libbinder.so android::IPThreadState::getAndExecuteCommand(this=0x000000762C03A8C0) + 156 <frameworks/native/

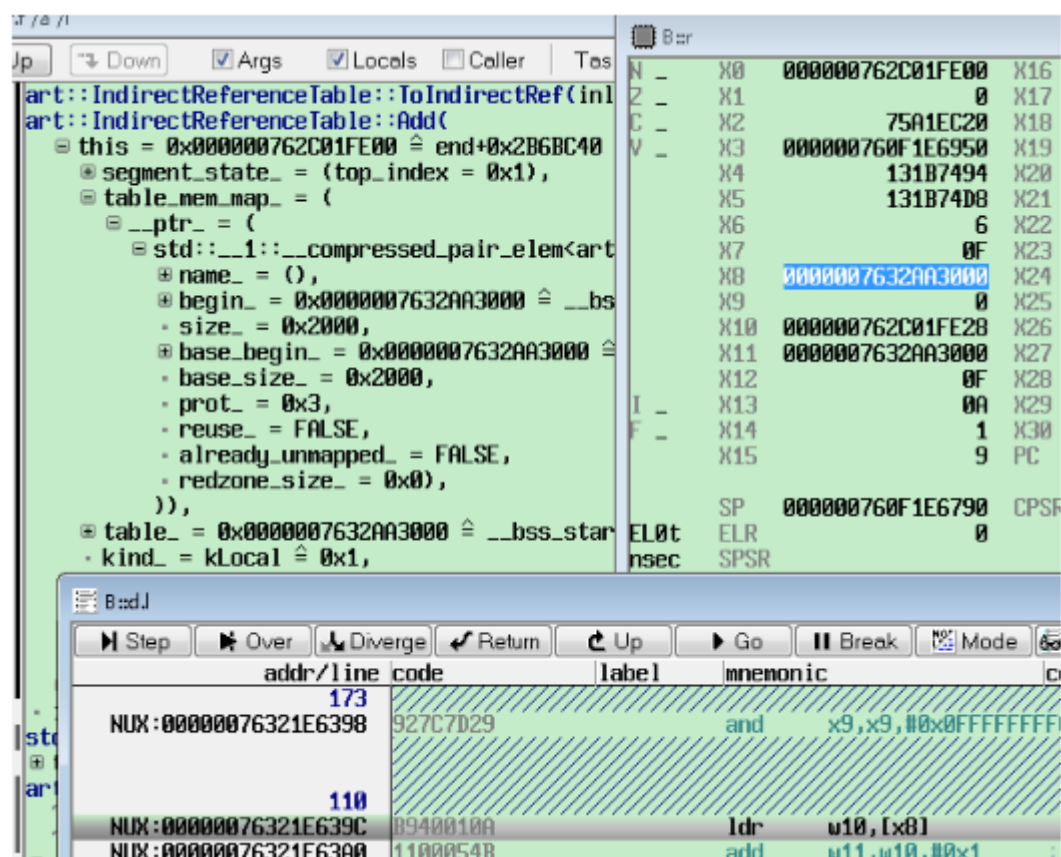

```

libs/binder/IPThreadState.cpp:458>
libbinder.so android::IPThreadState::joinThreadPool(this=0x000000762C03A8C0, isMain=false) + 60 <frameworks/
native/libs/binder/IPThreadState.cpp:538>
libbinder.so android::PoolThread::threadLoop(this=0x0000007628A319E0) + 28 <frameworks/native/libs/binder/Pro
cessState.cpp:63>
libutils.so android::Thread::_threadLoop(参数 1=0x0000007628A319E0) + 280
libandroid_runtime.so android::AndroidRuntime::javaThreadShell(args=0x0000007614FE2B60) + 140 <frameworks/bas
e/core/jni/AndroidRuntime.cpp:1273>
libc.so __pthread_start(arg=0x000000760F1E74F0) + 36 <bionic/libc/bionic/pthread_create.cpp:254>
libc.so __start_thread(fn=__pthread_start(), arg=0x000000760F1E74F0) + 68 <bionic/libc/bionic/clone.cpp:52>
== 栈结束 ==

```

挂在 ART 里，首先看下 IndirectReferenceTable 代码逻辑，不复杂。IndirectReferenceTable 核心是一个数组，可以添加和删除 obj。IndirectReferenceTable 有 3 种类型：global，local，weakglobal。

其中 global 和 weakglobal 各只有一个 IndirectReferenceTable 实例，而 local 是 per thread 的。出问题的是某个 thread 的 local IndirectReferenceTable 内存异常。我们仔细看结构体：



访问 0x7632AA3000 时 NE，看 PROCESS_MAPS：

```

7632aa2000-7632aa3000 rw-p 00010000 fc:00 3370 /system/lib64/libtombstoned_client.so
7632aa4000-7632aa5000 rw-p 00001000 00:04 27060917 /dev/ashmem/dalvik-indirect ref table (locals_)

```

就没有 0x7632AA3000 对应的内存。但结构体 base_begin_ = 0x7632AA3000。所以怀疑有人意外把 IndirectReferenceTable munmap 掉。

后面测试又打中好几例，有些是在 IndirectReferenceTable NE，有些是 art 其他内存被 munmap 引起的 NE。

== 异常报告 v3.2(仅供参考) ==

报告解读: MediaTek On-Line> Quick Start> NE/KE 分析报告解读> NE 分析报告

详细描述: 写数据 0xA646F808 到错误的地址(0xAB329000)，请结合崩溃线程调用栈检查相关代码

版本 : alps-mp-p0.mpl.tc1sp/userdebug build

发生时间: Thu Nov 8 04:22:59 CST 2018

命令行 : com.android.launcher3

pid : 7093, ppid: 6522 (zygote)

== 线程信息(共 38 个线程) ==

当前线程信息:

线程名: Jit thread pool worker thread 0, 可中断睡眠, tid: 7098

errno: 22

线程状态: 在调用 JNI 方法

本地调用栈:

libc.so memset() + 48 <bionic/libc/arch-arm/cortex-a7/bionic/memset.S:81>

libart-compiler.so art::HScheduler::Schedule(参数 1=0xA3FFF1C0, 参数 2=0xA646F790) + 550

libart-compiler.so art::HScheduler::Schedule(参数 2=0xA608C000) + 202

libart-compiler.so art::HInstructionScheduling::Run() + 216

libart-compiler.so 0xA45089C1(参数 1=0xA9FBDF80, 参数 2=0xA608C000) + 90

libart-compiler.so art::OptimizingCompiler::RunOptimizations(参数 1=0xA9FBDF80, 参数 2=0xA608C000, 参数 3=0xA608C100, 参数 4=0xA3FFF738, 参数 5=0xA3FFF4B0, 参数 6=0xA3FFF638) + 506

libart-compiler.so art::OptimizingCompiler::TryCompile(参数 1=0xA9FBDF80, 参数 2=0xA3FFF688, 参数 3=0xA3FFF660, 参数 4=0xA3FFF648, 参数 5=0xA3FFF738, 参数 6=0xABD3C5C8, 参数 7=0, 参数 8=0xA3FFF638) + 1606

libart-compiler.so art::OptimizingCompiler::JitCompile(参数 2=0xA4008000, 参数 4=0xABD3C5C8, 参数 5=0, 参数 6=0) + 634

libart-compiler.so art::jit::JitCompiler::CompileMethod(参数 2=0xABD3C5C8, 参数 3=0xA4008000, 参数 4=0) + 130

libart.so art::jit::Jit::CompileMethod(参数 2=0xABD3C5C8, 参数 3=0xA4008000, 参数 4=1) + 440

libart.so 0xA9C49A9D(参数 1=0x8EB769B0, 参数 2=0xA4008000) + 422

libart.so art::ThreadPoolWorker::Run(参数 1=0xAA00E9A0) + 44

libart.so art::ThreadPoolWorker::Callback(参数 1=0xAA00E9A0, 参数 2=art::ThreadPoolWorker::Callback()) + 94

libc.so __pthread_start(arg=0xA3FFF970) + 22 <bionic/libc/bionic/pthread_create.cpp:254>

libc.so __start_thread(fn=__pthread_start()) + 24 <bionic/libc/bionic/clone.cpp:52>

== 栈结束 ==

0xAB329000 对应的 PROCESS_MAPS:

ab328000-ab329000 rw-p 00000000 00:04 121878 /dev/ashmem/dalvik-CompilerMetadata (deleted)

ab32a000-ab348000 rw-p 00002000 00:04 121878 /dev/ashmem/dalvik-CompilerMetadata (deleted)

dalvik-CompilerMetadata 被 munmap 成 2 份了。

需要有个对策抓住凶手，观察到 munmap 的特征：

- munmap 通常是一个 page
- 基本都是落在 ART 申请的内存上
- 基本都存在：gralloc : Warning shared attribute region mapped at free. Unmapping 这样的 log

最后一点很重要，找到对应的代码：

```
if( hnd->attr_base != MAP_FAILED )
{
    ALOGW("Warning shared attribute region mapped at free. Unmapping");
    munmap( hnd->attr_base, PAGE_SIZE );
    hnd->attr_base = MAP_FAILED;
}
```

刚好也是一个 page，非常怀疑 gralloc 导致的 NE。口说无凭，根据特征设计 debugging 方案：

- libc hook munmap 函数
- kernel hook munmap 函数

libc 比较麻烦，最后考虑在 kernel do_munmap 函数里加代码拦截：

```
diff --git a/mm/mmap.c b/mm/mmap.c
index e875c11..fe0a403 100644
--- a/mm/mmap.c
+++ b/mm/mmap.c

@@ -2800,7 +2800,18 @@
     end = start + len;
     if (vma->vm_start >= end)
         return 0;

-
+     if (start == vma->vm_start
+         && len == PAGE_SIZE
+         && vma->vm_end == start + PAGE_SIZE * 2
+         && vma->vm_file
+         && vma->vm_file->f_path.dentry
+         && strstr(vma->vm_file->f_path.dentry->d_name.name
+                 , "dalvik-indirect ref table")) {
+         pr_info("yan: send sig 6, %s\n"
+                 , vma->vm_file->f_path.dentry->d_name.name);
+         show_stack(NULL, NULL);
+         send_sig(6, current, 0);
+     }
+     /*
+      * If we need to split any vma, do it now to save pain later.
+      */
```

经过几轮测试，终于有一例中枪，抓到 db 分析：

== 异常报告 v3.2(仅供参考) ==

报告解读: MediaTek On-Line> Quick Start> NE/KE 分析报告解读> NE 分析报告

详细描述: 程序主动调用 abort()

版本 : alps-mp-p0.mpl.tclsp/userdebug build

发生时间: Thu Nov 8 04:57:08 CST 2018

命令行 : com.android.systemui

pid : 6938, ppid: 13254 (zygote)

== 线程信息(共 59 个线程) ==

当前线程信息:

线程名: FinalizerDaemon, 可中断睡眠, tid: 6947

errno: 11

线程状态: 在调用 JNI 方法

Java 调用栈:

..... android.graphics.GraphicBuffer.nDestroyGraphicBuffer() <本地方法>

boot-framework.oat android.graphics.GraphicBuffer.finalize()

boot-core-libart.oat java.lang.Daemons\$FinalizerDaemon.doFinalize()

boot-core-libart.oat java.lang.Daemons\$FinalizerDaemon.runInternal()

boot-core-libart.oat java.lang.Daemons\$Daemon.run()

boot.oat java.lang.Thread.run()

== 栈结束 ==

本地调用栈:

libc.so munmap(参数 1=0xA41AA000, 参数 2=4096) + 12 <bionic/libc/arch-arm/syscalls/munmap.S:10>

gralloc.mt6750.so gralloc_buffer_attr_free(hnd=0x9CF278A0) + 44 <vendor/mediatek/proprietary/hardware/gpu_mali/mali_midgard/r26p0-01rel0/product/android/gralloc/src/gralloc_buffer_priv.cpp:135>

gralloc.mt6750.so mali_gralloc_reference_release(handle=0x9CF278A0, canFree=true) + 292 <vendor/mediatek/proprietary/hardware/gpu_mali/mali_midgard/r26p0-01rel0/product/android/gralloc/src/mali_gralloc_reference.cpp:144>

gralloc.mt6750.so mali_gralloc_release(device=0x86BD9E40, buffer=0x9CF278A0) + 6 <vendor/mediatek/proprietary/hardware/gpu_mali/mali_midgard/r26p0-01rel0/product/android/gralloc/src/mali_gralloc_public_interface.cpp:269>

android.hardware.graphics.mapper@2.0-impl.so 0x9664FEF1(参数 1=0x869AA480) + 10

android.hardware.graphics.mapper@2.0-impl.so 0x96650A35(参数 1=0x89A9E510, 参数 2=0x869C51D0) + 26

android.hardware.graphics.mapper@2.0.so 0xA666342D(参数 1=0x89A9E510, 参数 3=0x9CF278A0) + 84

libui.so android::Gralloc2::Mapper::freeBuffer(参数 1=0x9AD2DB08, 参数 2=0x9CF278A0) + 26

libui.so android::GraphicBufferMapper::freeBuffer(参数 1=0x9AD2DAD8, 参数 2=0) + 38

libui.so android::GraphicBuffer::free_handle(参数 1=0x9CF69B40) + 90

libui.so android::GraphicBuffer::~~GraphicBuffer(参数 1=0x9CF69B40) + 20

libutils.so android::RefBase::decStrong(this=0x9CF69B40, id=0x9CF2FE20) + 66 <system/core/libutils/RefBase.cpp:434>

libandroid_runtime.so android::android_graphics_GraphicBuffer_destroy(参数 1=0x9CF20080, 参数 2=0x89A9E58C, 参数 3=0x9CF2FE20, 参数 4=0) + 12 <frameworks/native/libs/ui/include/ui/ANativeObjectBase.h:47>

system@framework@boot-framework.oat android.graphics.GraphicBuffer.nDestroyGraphicBuffer(参数 2=0x13649CE8) +

```

102 <本地方法>

system@framework@boot-framework.oat android.graphics.GraphicBuffer.finalize(参数 2=0x13649CE8) + 60
system@framework@boot-core-libart.oat java.lang.Daemons$FinalizerDaemon.doFinalize(参数 2=0x6F97DD80, 参数 3=0x13427158) + 86

system@framework@boot-core-libart.oat java.lang.Daemons$FinalizerDaemon.runInternal(参数 2=0x6F97DD80) + 466
system@framework@boot-core-libart.oat java.lang.Daemons$Daemon.run(参数 2=0x6F97DD80) + 66
system@framework@boot.oat java.lang.Thread.run(参数 1=0) + 64

libart.so 0xA2D48731() + 68

libart.so art_quick_invoke_stub(参数 1=0x6F9008A0, 参数 2=0x89A9E87C, 参数 3=4, 参数 4=0xA26CAA00) + 224
libart.so art::ArtMethod::Invoke(参数 1=0x6F9008A0, 参数 2=0xA26CAA00, 参数 3=0x89A9E87C, 参数 4=4, 参数 5=0x89A9E860, 参数 6=0x70D7E975) + 136

libart.so 0xA2C82AB1(参数 1=0x89A9E914, 参数 2=0x6F9008A0, 参数 3=0x89A9E86C, 参数 4=0x89A9E860) + 52
libart.so art::InvokeVirtualOrInterfaceWithJValues(参数 1=0x89A9E900, 参数 2=0x89A9E914, 参数 3=5, 参数 4=0x6F9008A0, 参数 5=0) + 320

libart.so art::Thread::CreateCallback(参数 1=0xA26CAA00, 参数 2=art::Thread::CreateCallback()) + 866
libc.so __pthread_start(arg=0x89A9E970) + 22 <bionic/libc/bionic/pthread_create.cpp:254>
libc.so __start_thread(fn=__pthread_start()) + 24 <bionic/libc/bionic/clone.cpp:52>

== 栈结束 ==

```

同时 kernel log 有打印:

```

[39008.320416] (1)[6947:FinalizerDaemon]mmap: yan: send sig 6, dev/ashmem/dalvik-indirect ref table
[39008.320434] (1)[6947:FinalizerDaemon]Backtrace:
[39008.320465] (1)[6947:FinalizerDaemon][] (dump_backtrace) from [] (show_stack+0x18/0x1c)
[39008.320483] (1)[6947:FinalizerDaemon] r6:a41aa000 r5:a41ab000 r4:d31f9588 r3:dc8cb077
[39008.320519] (1)[6947:FinalizerDaemon][] (show_stack) from [] (do_munmap+0x43c/0x46c)
[39008.320537] (1)[6947:FinalizerDaemon][] (do_munmap) from [] (Sys_munmap+0x48/0x5c)
[39008.320545] (1)[6947:FinalizerDaemon] r10:00000800 r9:d6caa000 r8:c0207e64 r7:a41aa000
[39008.320580] (1)[6947:FinalizerDaemon][] (Sys_munmap) from [] (__sys_trace_return+0x0/0x2c)
[39008.320605] -(1)[6947:FinalizerDaemon][name:mtprof&][signal][6947:FinalizerDaemon] send death sig 6 to [6947:FinalizerDaemon:R]

```

和之前猜测 gralloc 是凶手吻合。

根本原因

gralloc 错误将别人的 memory 给 munmap 掉

解决方法

修正 gralloc 代码

结语

Android 非常庞大且更新快速，基本都是在解决问题过程中学习的，比如 IndirectReferenceTable

问题背景

P 版本做 reboot 测试，概率性发生 NE，陆续收到 8 台同样问题的手机，问题严重。

分析过程

利用工具 SpOfflineDebugSuite 分析，产生分析报告如下：

== 异常报告 v2.1(仅供参考) ==

报告解读: MediaTek On-Line> Quick Start> NE/KE 分析报告解读> NE 分析报告

详细描述: 程序正常，有可能是硬件问题(检查 PCB 是否经过线路仿真)或外部触发的信号导致

版本 : alps-mp-p0.mp3.tc19sp/user build

发生时间: Fri Jan 4 08:45:14 CST 2019

命令行 : system_server

pid : 1088, ppid: 560 (zygote64)

== 线程信息(共 179 个线程) ==

当前线程信息:

线程名: system_server, 可中断睡眠, tid: 2896

本地调用栈:

libc.so _exit_with_stack_teardown() + 8 <bionic/libc/arch-arm64/bionic/_exit_with_stack_teardown.S:37>

libc.so pthread_exit(return_value=0) + 244 <bionic/libc/bionic/pthread_exit.cpp:129>

== 栈结束 ==

对应汇编指令:

行号 地址 指令 提示

bionic/libc/arch-arm64/bionic/_exit_with_stack_teardown.S

37 : 0000007647F61A18: MOV X0, #0 ; 线程停止在这里

当时的寄存器值:

X0: 0000000000000000, X1: 0000000000106000, X2: 000000759E538468, X3: 0000000000000008

X4: 0000000000000000, X5: 00000075A7E3BCF0, X6: 0000000000000000, X7: 0000000100000002

X8: 00000000000000D7, X9: 1A18E1D44B69BD2C, X10: FFFFFFFF87FFFBFF, X11: 0000000000000001

X12: 0000000000000001, X13: 000000764803E050, X14: 00000000FFFFFFF, X15: AAAAAAAAAAAAAAB

X16: 0000007648032208, X17: 0000007647FB36B8, X18: 000000764803E000, X19: 000000759E5384F0

X20: 0000000000000000, X21: 000000759E5384F0, X22: 00000440000004E8, X23: 0000000000000004

X24: 000000759E538570, X25: 000000759E433000, X26: 00000075A3D78588, X27: 00000075C6A39438

X28: 00000075C6BEE048, X29: 000000759E538490, X30: 0000007647FC69E0, SP: 000000759E538460

PC: 0000007647F61A18

非常奇怪 NE 的位置并不会发生 sig 11，而且执行到这里的时候 stack 已经 munmap 了。按下面思路分析：

1. 怀疑 HW 不良：不可能，项目刚开始规模测试，复现到的问题现象一致，都是 NE。
2. 怀疑 kernel 有故意发送 sig 11：和客户一起排查，没有查到相关的代码。
3. 搜索 2896 线程之前做过什么事情：查找 SYS_ANDROID_LOG，发现如下 log：


```

01-04 08:45:11.758 1088 2896 D VAPIBlackWhiteInfoServer: get Secure Config!
01-04 08:45:11.759 1088 2896 E ActivityThread: Failed to find provider info for xxx
01-04 08:45:11.759 1088 2896 D VAPIBlackWhiteInfoServer: cursor is null, lock failed, continue checking for update!
01-04 08:45:11.759 1088 2896 W FeatureService: VAPI getSecureConfigToVAPITable failed! times:6

```

- 搜索其他几个 db 也都是这样的 log，直接找对应的代码，发现 new thread 之后执行一段代码然后就退出了，做的事情也很简单，没问题。

4. 在 bsp log 里搜索 2896，发现关键点：

```

5. -1256 [002] .... 26.173165: sched_fork_time: comm=AudioService pid=1256 child_comm=AudioService child_pid=2896 fork_time=52769 us
    -1256 [002] .... 26.173542: sched_fork_time: comm=AudioService pid=1256 child_comm=AudioService child_pid=2897 fork_time=49462 us
    -1093 [000] d..1 26.199979: signal_generate: sig=33 errno=0 code=-6 comm=Thread-30 pid=2896 grp=0 res=0
    -2896 [002] d..1 26.200012: signal_deliver: sig=33 errno=0 code=-6 sa_handler=764c88b7e8 sa_flags=18000004
    -2896 [002] d..1 26.200058: signal_generate: sig=11 errno=0 code=128 comm=Thread-30 pid=2896 grp=0 res=0
    -2896 [002] d..1 26.200062: signal_deliver: sig=11 errno=0 code=128 sa_handler=55bddb2948 sa_flags=18000004
    -2896 [002] d..1 26.200103: signal_generate: sig=11 errno=0 code=128 comm=Thread-30 pid=2896 grp=0 res=0
    -2896 [002] d..1 26.200105: signal_deliver: sig=11 errno=0 code=128 sa_handler=0 sa_flags=18000004

```

发现 sig 11 前有 sig 33 送过来，而 sig 33 是用来 dump backtrace 的，具体看 libbacktrace 代码。

回到 coredump，sig 33 一般是 Signal Catcher 发出

线程名: Signal Catcher, 不可中断睡眠, tid: 1093

线程状态: 在调用 JNI 方法

本地调用栈:

libc.so syscall(参数 1=98, 参数 2=0x000000759E64945C, 参数 3=137, 参数 4=2, 参数 5=0x00000075C08AEF18, 参数 6=0, 参数 7=0x00000000FFFFFFFF) + 28 <bionic/libc/arch-arm64/bionic/syscall.S:41>

libc.so __futex(ftx=0x000000759E64945C, value=2, timeout=0x00000075C08AEF18) + 40 <bionic/libc/private/bionic_futex.h:45>

libc.so FutexWithTimeout() + 72 <bionic/libc/bionic/bionic_futex.cpp:58>

libc.so __futex_wait_ex(ftx=0x000000759E64945C, value=2, use_realtime_clock=false, abs_timeout=0x00000075C08AEF18) + 140 <bionic/libc/bionic/bionic_futex.cpp:63>

libc.so __pthread_cond_timedwait(cond=0x000000759E64945C, abs_timeout_or_null=0x00000075C08AEF18) + 84 <bionic/libc/bionic/pthread_cond.cpp:182>

```

libc.so pthread_cond_timedwait(cond_interface=0x000000759E64945C, mutex=0x000000759E649434, abstime=0x00000075C08AEF18) + 120 <bionic/libc/bionic/pthread_cond.cpp:209>
libbacktrace.so ThreadEntry::Wait(this=0x000000759E649400, value=1) + 112 <system/core/libbacktrace/ThreadEntry.cpp:107>
libbacktrace.so BacktraceCurrent::UnwindThread(this=0x00000075C04E65A0, num_ignore_frames=0) + 324 <system/core/libbacktrace/BacktraceCurrent.cpp:204>
libart.so art::DumpNativeStack(参数 1=0x00000075C08AF2B0, 参数 2=2896, 参数 3=0, 参数 4=0x00000075C6B43E8A, 参数 5=0, 参数 6=0, 参数 7=1) + 220 <art/runtime/native_stack_dump.cc:305>
libart.so art::DumpUnattachedThread(os=0x00000075C08AF2B0) + 68 <art/runtime/thread_list.cc:168>
libart.so art::ThreadList::DumpUnattachedThreads(this=0x00000075C6D22000, os=0x00000075C08AF2B0, dump_native_stack=true) + 340 <art/runtime/thread_list.cc:191>
libart.so art::ThreadList::DumpForSigQuit(this=0x00000075C6D22000, os=0x00000075C08AF2B0) + 900 <art/runtime/thread_list.cc:158>
libart.so art::Runtime::DumpForSigQuit(this=0x00000075C6C4B700, os=0x00000075C08AF2B0) + 188 <art/runtime/runtime.cc:1828>
libart.so art::SignalCatcher::HandleSigQuit(this=0x00000075BD0130A0) + 1372 <art/runtime/signal_catcher.cc:196>
libart.so art::SignalCatcher::Run(arg=0x00000075BD0130A0) + 256 <art/runtime/signal_catcher.cc:264>
libc.so __pthread_start(arg=0x00000075C08AF4F0) + 36 <bionic/libc/bionic/pthread_create.cpp:254>
libc.so __start_thread(fn=__pthread_start(), arg=0x00000075C08AF4F0) + 68 <bionic/libc/bionic/clone.cpp:52>
== 栈结束 ==

```

明显看到 DumpNativeStack 正在 dump 2896 这个 tid 的 stack，正等待 sig 33 响应呢。用 trace32 分析 BacktraceCurrent::UnwindThread 的 this，发现还在等待 2896 signal handler 执行，也就是说 2896 的 sig 33 handler 还没被执行。

理论上 pthread_exit 不应该响应任何 sig，从代码上也是这样：

```

121  if (thread->mmap_size != 0) {
122      // We need to free mapped space for detached threads when they exit.
123      // That's not something we can do in C.
124
125      // We don't want to take a signal after we've unmapped the stack.
126      // That's one last thing we can do before dropping to assembler.
127      ScopedSignalBlocker ssb;
128      __pthread_unmap_tls(thread);
129      _exit_with_stack_tear_down(thread->attr.stack_base, thread->mmap_size);
130  }

```

ssb 会阻止所有信号进来，但为何在 _exit_with_stack_tear_down 还能收到 sig 33 呢？是否是 kernel 哪些地方会 unmask block signal？检查代码也没发现有。

刚好 db 里有 THREAD_STATE，里面有：

SigQ: 1/14268
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: ffffffff87ffbfaff

可以看到 SigBlk 里的 bit33 被清除了，还有其他 bit 被清除了。这就诡异了，再仔细检查代码看 ScopedSignalBlocker，这个类包装了 sigprocmask64，看 sigprocmask64，终于发现问题：

```
int sigprocmask64(int how,
                  const sigset64_t* new_set,
                  sigset64_t* old_set) __attribute__((__noinline__))
{
    sigset64_t mutable_new_set;
    sigset64_t* mutable_new_set_ptr = nullptr;
    if (new_set) {
        mutable_new_set = filter_reserved_signals(*new_set);
        mutable_new_set_ptr = &mutable_new_set;
    }

    // Realtime signals reserved for internal use:
    // 32 (__SIGRTMIN + 0)    POSIX timers
    // 33 (__SIGRTMIN + 1)    libbacktrace
    // 34 (__SIGRTMIN + 2)    libcore
    // 35 (__SIGRTMIN + 3)    debuggerd -b
    //
    // If you change this, also change __ndk_legacy__libc_current_sigrtmin
    // in <android/legacy_signal_inlines.h> to match.

#define __SIGRT_RESERVED 4
    static inline __always_inline sigset64_t filter_reserved_signals(sigset64_t sigset)
    {
        for (int signo = __SIGRTMIN; signo < __SIGRTMIN + __SIGRT_RESERVED; ++signo) {
            sigdelset64(&sigset, signo);
        }
    }
}
```

从代码看会故意保留 sig 32, 33, 34, 35，这就是导致问题的根本原因。

解决方法

需要 google fix。临时方案：

在文件头部申明：

```
extern "C" int __rt_sigprocmask(int, const kernel_sigset_t*, kernel_sigset_t*, size_t);
```

然后

```
// ScopedSignalBlocker ssb; // delete this line
```

```
// add this block
```

```
{
    sigset64_t new_set, old_set;

    sigfillset64(&new_set);
    __rt_sigprocmask(SIG_SETMASK, &new_set, &old_set, sizeof(new_set));
}
```

结语

从 debugging 中学习 libbacktrace，linux signal 机制。

遗留一个问题给大家思考：sig 33 后面的 sig 11 从哪里来？是从 page fault 来吗？

10. camerahalserver NE -- 访问数组越界

问题背景

稳定性测试打出一例 lib3a.flash.so NE

分析过程

1. camera owner 根据如下 log 中的 backtrace 使用 addr2line 还原调用栈:

```
67081 02-02 23:34:03.415 17605 17605 F DEBUG      :

67082 02-02 23:34:03.415 17605 17605 F DEBUG      : backtrace:

67083 02-02 23:34:03.415 17605 17605 F DEBUG      :          #00 pc 0000000000010464  /vendor/lib64/lib3a.flash.so
(NS3A::FlashAlgm::ForegroundSegment(double*, int, double*, double*, double*))+380)

67084 02-02 23:34:03.415 17605 17605 F DEBUG      :          #01 pc 000000000001105c  /vendor/lib64/lib3a.flash.so
(NS3A::FlashAlgm::BuildFbTable()+356)

67085 02-02 23:34:03.415 17605 17605 F DEBUG      :          #02 pc 00000000000146a0  /vendor/lib64/lib3a.flash.so
(NS3A::FlashAlgm::Estimate(NS3A::FlashAlgExpPara*, NS3A::FlashAlgFacePos*, int*))+168)

67086 02-02 23:34:03.415 17605 17605 F DEBUG      :          #03 pc
00000000001bab74  /vendor/lib64/libcam.hal3a.v3.so (FlashMgrM::pfRun(FlashExePara*, FlashExeRep*))+476)

67087 02-02 23:34:03.415 17605 17605 F DEBUG      :          #04 pc
00000000001bb918  /vendor/lib64/libcam.hal3a.v3.so (FlashMgrM::doPfOneFrameNormal(FlashExePara*,
FlashExeRep*))+632)

67088 02-02 23:34:03.415 17605 17605 F DEBUG      :          #05 pc
00000000001bb658  /vendor/lib64/libcam.hal3a.v3.so (FlashMgrM::doPfOneFrame(FlashExePara*, FlashExeRep*))+1256)

67089 02-02 23:34:03.415 17605 17605 F DEBUG      :          #06 pc
0000000000190c7c  /vendor/lib64/libcam.hal3a.v3.so (Task3AFlashBackImp::run(int, NS3Av3::TaskData const&))+2708)

67090 02-02 23:34:03.415 17605 17605 F DEBUG      :          #07 pc
0000000000195934  /vendor/lib64/libcam.hal3a.v3.so (TaskMgrImp::execute(NS3Av3::TASK_UPDATE))+1148)

67091 02-02 23:34:03.415 17605 17605 F DEBUG      :          #08 pc
0000000000b17c0  /vendor/lib64/libcam.hal3a.v3.so (Hal3ARawImp::postCommand(NS3Av3::ECmd_T,
NS3Av3::ParamIspProfile_T const*))+3800)

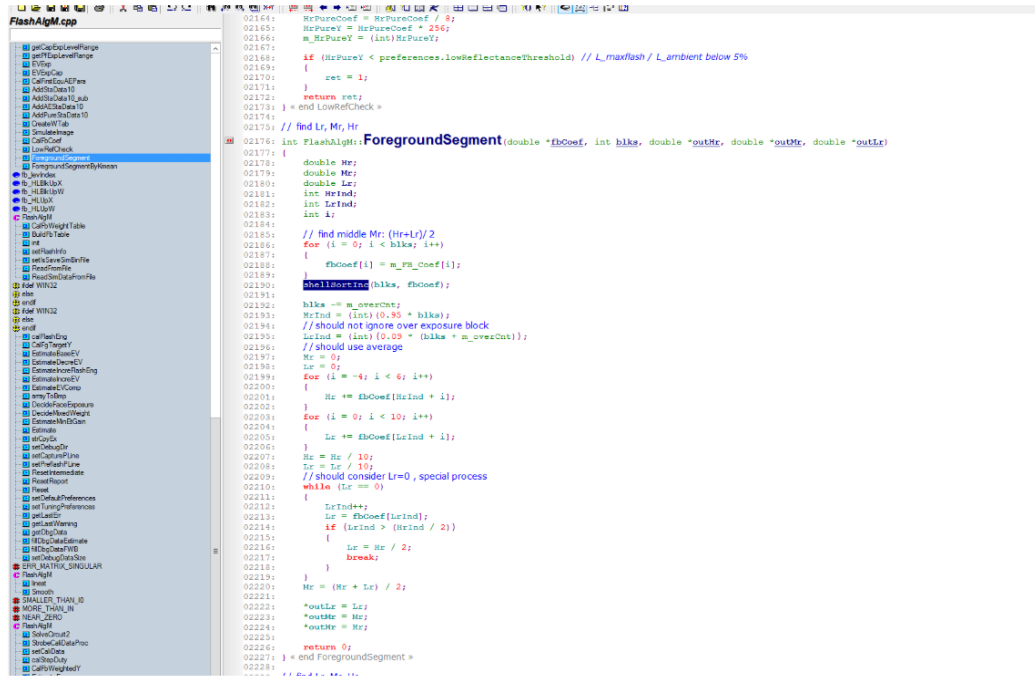
67092 02-02 23:34:03.415 17605 17605 F DEBUG      :          #09 pc
00000000001f1204  /vendor/lib64/libcam.hal3a.v3.so (NS3Av3::Hal3AFlowCtrl::postCommand(NS3Av3::ECmd_T,
NS3Av3::ParamIspProfile_T const*))+1380)
```

//对应的代码如下:

//从 foregroundsegment 分析并没有看到有特殊使用 memory 的地方, 怀疑是别的地方踩坏了 flash algo 的 memory

//对应的代码如下:

//从 foregroundsegment 分析并没有看到有特殊使用 memory 的地方, 怀疑是别的地方踩坏了 flash algo 的 memory



2. 请 camera owner 提供 NE db, 利用工具 SpOfflineDebugSuite 分析, 产生分析报告如下:

报告解读: MediaTek On-Line> Quick Start> NE/KE 分析报告解读> NE 分析报告

详细描述: 从错误的地址(0x0000007CC1A00000)读数据, 请结合崩溃线程调用栈检查相关代码

版本 : alps-mp-p0.mp3/user build

发生时间: Thu Jan 31 00:26:18 CST 2019

命令行 : /vendor/bin/hw/camerahalservice

pid : 645, ppid: 1 (init)

== 线程信息(共 157 个线程) ==

当前线程信息:

线程名: camerahalservice, 可中断睡眠, tid: 11297

本地调用栈:

lib3a.flash.so NS3A::FlashAlgm::ForegroundSegment(this=FlashAlgmDev::getInstance(void)::singleton, fbCoef=0x0000007CC19FFD80, blks=80, outHr=0x0000007CC27640F0, outMr=0x0000007CC27640E8, outLr=0x0000007CC27640E0) + 380 <vendor/mediatek/proprietary/hardware/libcamera_3a/libflash/mt6771/flash/FlashAlgm.cpp:2168>

lib3a.flash.so NS3A::FlashAlgm::BuildFbTable(this=FlashAlgmDev::getInstance(void)::singleton) + 356 <vendor/mediatek/proprietary/hardware/libcamera_3a/libflash/mt6771/flash/FlashAlgm.cpp:2558>

lib3a.flash.so NS3A::FlashAlgm::Estimate(this=FlashAlgmDev::getInstance(void)::singleton, exp=0x0000007CD8921D38, pFaceInfo=0x0000007CC2764B88, isLowRef=0x0000007CD8921E64) + 168 <vendor/mediatek/proprietary/hardware/libcamera_3a/libflash/mt6771/flash/FlashAlgm.cpp:3973>

libcam.hal3a.v3.so FlashMgrM::pfRun(参数 1=0x0000007CD8921CF0, 参数 2=0x0000007CC2769DC8, 参数 3=0x0000007CC276DFC8) + 476

libcam.hal3a.v3.so FlashMgrM::doPpOneFrameNormal(参数 1=0x0000007CD8921CF0, 参数 2=0x0000007CC2769DC8, 参数 3=0x0000007CC276DFC8) + 632

```

libcam.hal3a.v3.so FlashMgrM::doPfOneFrame() + 1256

libcam.hal3a.v3.so Task3AFlashBackImp::run(参数 1=0x0000007CC230F260, 参数 2=0, 参数 3=0x0000007CC25F2338) + 2708

libcam.hal3a.v3.so TaskMgrImp::execute(参数 1=0x0000007CC25F2280, 参数 2=0) + 1148

libcam.hal3a.v3.so Hal3ARawImp::postCommand(参数 1=0x0000007CD86E4D00, 参数 2=8, 参数 3=0x0000007CC272F00) + 3800

libcam.hal3a.v3.so NS3Av3::Hal3AFlowCtrl::postCommand(参数 1=0x0000007CD8933878, 参数 2=8, 参数 3=0x0000007CC27731E0) + 1380

libcam.hal3a.v3.so NS3Av3::Hal3AFlowCtrl::doUpdateCmd(参数 1=0x0000007CD8933878, 参数 2=0x0000007CC27731E0) + 660

libcam.hal3a.v3.so Thread3AImp::onThreadLoop(参数 1=0x0000007CDE6281C0) + 1056

libcam.hal3a.v3.so std::__1::__thread_proxy<__1::tuple< __1::unique_ptr<__1::__thread_struct, __1::default_delete<?>>, (?)*, Thread3AImp*>>>(参数 1=0x0000007CDE6DBE00) + 40

libc.so __pthread_start(arg=0x0000007CC27734F0) + 36 <bionic/libc/bionic/pthread_create.cpp:254>

libc.so __start_thread(fn=__pthread_start(), arg=0x0000007CC27734F0) + 68 <bionic/libc/bionic/clone.cpp:52>

== 栈结束 ==

对应汇编指令:

行号 地址 指令 提示

vendor/mediatek/proprietary/hardware/libcamera_3a/libflash/mt6771/flash/FlashAlgM.cpp

2168: 0000007CC65B5464: LDP D3, D4, [X11, #0x20] ; 线程停止在这里

```

vendor/mediatek/proprietary/hardware/libcamera_3a/libflash/mt6771/flash/FlashAlgM.cpp:2168 行在 ForegroundSegment 函数里, 并且用 trace32 打开看到 2168 行的汇编代码应该是对 x11 附近的数据循环读取:

```

2168 NUX:0000007CC65B5410 ldp d2,d4,[x11,#-0x20] ; d2,d4,[x11,#-32]
2168 NUX:0000007CC65B5414 fadd d0,d2,d0

2168 NUX:0000007CC65B5428 ldp d2,d4,[x11,#-0x10] ; d2,d4,[x11,#-16]
2168 NUX:0000007CC65B542C fadd d0,d0,d2

2168 NUX:0000007CC65B5440 ldp d2,d4,[x11]
2168 NUX:0000007CC65B5444 fadd d0,d0,d2

2168 NUX:0000007CC65B5450 fadd d0,d0,d4
2168 NUX:0000007CC65B5454 ldp d3,d4,[x11,#0x10] ; d3,d4,[x11,#16]
2175 NUX:0000007CC65B5458 fdiv d1,d1,d2

2168 NUX:0000007CC65B545C fadd d0,d0,d3
2168 NUX:0000007CC65B5460 fadd d0,d0,d4
2168 NUX:0000007CC65B5464 ldp d3,d4,[x11,#0x20] ; d3,d4,[x11,#32]

```

明显与上面的 source code 不对应

X11: 0000007CC19FFFE0, dump data 看到这种明显是访问的地址超出了内存块 size:

```

NUD:0000007CC19FFFC0 2320CC00 3FA6F5D6 D9D33238 3FA71E2E
NUD:0000007CC19FFFD0 4B595D90 3FA73B1D 5362DB97 3FA766EA
0501B182 3FA76C1C E1B6419E 3FA7A94B
NUD:0000007CC19FFFE0 865A84BF 3FA7CAFO 9251D438 3FA7F5A8
NUD:0000007CC1A00000 ?????????? ?????????? ??????????
NUD:0000007CC1A00010 ?????????? ?????????? ??????????

```


请 camera owner 找到新版的 source code，看起来代码是可以对应上了：

```
2164     Hr=0;
2165     Lr=0;
2166     for(i=-4;i<6;i++)
2167     {
2168         Hr += fbCoef[HrInd+i];
2169     }
```

很明显是对 fbCoef 数组访问越界，并非踩内存

根本原因

对 fbCoef 数组访问越界

解决方法

审查代码逻辑，对 fbCoef 数组正确操作

11.lock 不是原子导致了 NE

问题背景

0 版本做稳定性测试，极低概率性发生 NE，问题持续了半年，问题严重。

分析过程

利用工具 SpOfflineDebugSuite 分析，产生分析报告如下：

== 异常报告 v2.1 (仅供参考) ==

报告解读: MediaTek On-Line> Quick Start> NE/KE 分析报告解读> NE 分析报告

详细描述: 从错误的地址 (0x00000076CDA26574) 读数据，请结合崩溃线程调用栈检查相关代码

版本 : alps-mp-o1.mp6/user build

发生时间: Thu Jan 3 05:02:00 CST 2019

命令行 : system_server

pid : 1039, ppid: 567 (zygote64)

== 线程信息 (共 195 个线程) ==

当前线程信息:

线程名: main, 被跟踪, tid: 1039 (主线程)

线程状态: 在调用 JNI 方法

Java 调用栈:

..... java.lang.Thread.nativeCreate() <本地方法>

/system/framework/arm64/boot.vdex java.lang.Thread.start()

/system/framework/oat/arm64/services.vdex com.android.server.DropBoxManagerService\$1.onReceive()

/system/framework/oat/arm64/services.vdex com.android.server.DropBoxManagerService\$4.onChange()

/system/framework/arm64/boot-framework.vdex android.database.ContentObserver.onChange()

```

/system/framework/arm64/boot-framework.vdex android.database.ContentObserver.onChange()
/system/framework/arm64/boot-framework.vdex android.database.ContentObserver$NotificationRunnable.run()
/system/framework/arm64/boot-framework.vdex android.os.Handler.dispatchMessage()
/system/framework/arm64/boot-framework.vdex android.os.Looper.loop()
/system/framework/oat/arm64/services.vdex com.android.server.SystemServer.run()
/system/framework/oat/arm64/services.vdex com.android.server.SystemServer.main()
..... java.lang.reflect.Method.invoke() <本地方法>
/system/framework/arm64/boot-framework.vdex com.android.internal.os.RuntimeInit$MethodAndArgsCaller.run()
/system/framework/arm64/boot-framework.vdex com.android.internal.os.ZygoteInit.main()
== 栈结束 ==
本地调用栈:
libc.so Lock::unlock() + 20 <bionic/libc/private/bionic_lock.h:74>
libc.so pthread_create(thread_out=0x0000007FC2B3ACB8, attr=0x0000007FC2B3ACF0, start_routine=art::Thread::CreateCallback(), arg=0x00000076D79B9E00) + 872 <bionic/libc/bionic/pthread_create.cpp:310>
libart.so art::Thread::CreateNativeThread(参数 1=0x00000076F6678B20) + 608
libart.so 0x00000076F5FD7210(参数 1=0x00000076F6678B20, 参数 2=0x0000007FC2B3ADE4) + 156
system@framework@boot.oat java.lang.Thread.nativeCreate(参数 2=0x0000000013BC1738, 参数 3=0, 参数 4=0) + 188 <本地方法>
system@framework@boot.oat java.lang.Thread.start(参数 2=0x0000000013BC1738) + 148
services.odex com.android.server.DropBoxManagerService$1.onReceive(参数 2=0x000000001451FEA0, 参数 3=0x0000000012C40098, 参数 4=0) + 184
services.odex com.android.server.DropBoxManagerService$4.onChange(参数 2=0x00000000146E9260) + 140
system@framework@boot-framework.oat android.database.ContentObserver.onChange(参数 2=0x00000000146E9260, 参数 4=0x0000000013AC31C0) + 48
system@framework@boot-framework.oat android.database.ContentObserver.onChange(参数 2=0x00000000146E9260, 参数 4=0x0000000013AC31C0, 参数 5=0) + 48
system@framework@boot-framework.oat android.database.ContentObserver$NotificationRunnable.run(参数 2=0x0000000013BC0B50) + 84
system@framework@boot-framework.oat android.os.Handler.dispatchMessage(参数 2=0x00000000146E9278, 参数 3=0x0000000015B2E988) + 76
system@framework@boot-framework.oat android.os.Looper.loop() + 1452
services.odex com.android.server.SystemServer.run(参数 2=0x0000000012C40130) + 3052
services.odex com.android.server.SystemServer.main() + 244
libart.so 0x00000076F613DFF0(参数 1=0x0000000097F00E60, 参数 2=0x0000007FC2B3B4D8, 参数 3=4, 参数 4=0x00000076F6660E00, 参数 5=0x0000007FC2B3B4A8, 参数 6=0x00000076DF45EB24) + 604
libart.so art::ArtMethod::Invoke(参数 1=0x0000000097F00E60, 参数 2=0x00000076F6660E00, 参数 3=0x0000007FC2B3B4D8) + 264
libart.so 0x00000076F60619C4(参数 1=0x0000007FC2B3B5B0, 参数 2=0x0000000097F00E60, 参数 3=0x0000007FC2B3B4C0, 参数 4=0x0000007FC2B3B4A8, 参数 5=0x00000076DF45EB24) + 100
libart.so art::InvokeMethod(参数 1=0x0000007FC2B3B5B0, 参数 5=1) + 1456
libart.so 0x00000076F5FE6CBC(参数 1=0x00000076F6678B20, 参数 2=0x0000007FC2B3B604) + 48
system@framework@boot.oat java.lang.reflect.Method.invoke(参数 2=0x0000000012C40030, 参数 3=0) + 180 <本地方法>
system@framework@boot-framework.oat com.android.internal.os.RuntimeInit$MethodAndArgsCaller.run(参数 2=0x000000012C401D0) + 132

```

```

system@framework@boot-framework.oat com.android.internal.os.ZygoteInit.main() + 2952

libart.so 0x00000076F613DFF0(参数 1=0x00000000712D7240, 参数 2=0x0000007FC2B3BA20, 参数 3=4, 参数 4=0x00000076F6
660E00, 参数 5=0x0000007FC2B3BA00, 参数 6=0x00000076F4CBE5D9) + 604

libart.so art::ArtMethod::Invoke(参数 1=0x00000000712D7240, 参数 2=0x00000076F6660E00, 参数 3=0x0000007FC2B3BA2
0) + 264

libart.so 0x00000076F60619C4(参数 1=0x0000007FC2B3BB08, 参数 2=0x00000000712D7240, 参数 3=0x0000007FC2B3BA08,
参数 4=0x0000007FC2B3BA00, 参数 5=0x00000076F4CBE5D9) + 100

libart.so art::InvokeWithVarArgs(参数 1=0x0000007FC2B3BB08, 参数 2=0, 参数 3=0x00000000712D7240, 参数 4=0x000000
7FC2B3BAD0) + 412

libart.so 0x00000076F5F67D20(参数 4=0x0000007FC2B3BC60) + 612

libandroid_runtime.so 0x0000007779F76E20(参数 1=0x00000076F6678B20, 参数 2=105, 参数 4=17) + 120

libandroid_runtime.so android::AndroidRuntime::start(参数 1=0x0000007FC2B3BE68, 参数 3=0x0000007FC2B3BE30, 参数
4=0) + 964

app_process64 main(argc=6, argv=0x0000007FC2B3CFC8) + 1328 <frameworks/base/cmds/app_process/app_main.cpp:0>

libc.so __libc_init(raw_args=0x0000007FC2B3CFC0, onexit=0, slingshot=main(), structors=0x0000007FC2B3CFA8) +
88 <bionic/libc/bionic/libc_init_dynamic.cpp:124>

app_process64 _start_main() + 80

== 栈结束 ==

```

对应代码如下：

```

int pthread_create(pthread_t* thread_out, pthread_attr_t const* attr, void* (*start_routine)(void*), void* ar
g) {
    ErrnoRestorer errno_restorer;

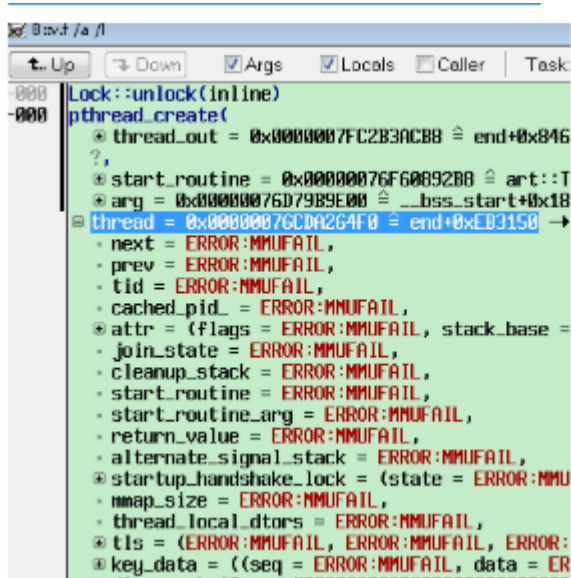
    .....

    // Publish the pthread_t and unlock the mutex to let the new thread start running.
    *thread_out = __pthread_internal_add(thread);
    thread->startup_handshake_lock.unlock(); /* crash here */

    return 0;
}

```

非常奇怪发现整个 thread 都不见了，用 trace32 看：



难道还没 unlock 线程就退出了？看代码子线程必须要等父线程 unlock 才能往下跑。

```
static int __pthread_start(void* arg) {
    pthread_internal_t* thread = reinterpret_cast<pthread_internal_t*>(arg);

    // Wait for our creating thread to release us. This lets it have time to
    // notify gdb about this thread before we start doing anything.
    // This also provides the memory barrier needed to ensure that all memory
    // accesses previously made by the creating thread are visible to us.
    thread->startup_handshake_lock.lock();

    __init_alternate_signal_stack(thread);

    void* result = thread->start_routine(thread->start_routine_arg);
    pthread_exit(result);

    return 0;
}
```

如果不是子线程偷跑，那么又是谁把 thread 给 munmap 掉？如果是意外被 munmap 掉，那么存在如下问题：

- g_thread_list 还残留有 thread 信息，但 coredump 来看 g_thread_list 正常，没有断裂，看起来子线程至少调用过 __pthread_internal_remove 函数。

当时想了一个 debugging 方法，在 thread 结构体增加成员 int isdone；在 unlock 之后才设置为 1，在 __pthread_internal_remove() 判断 isdone 是否为 1，如果为 0 就 abort。结果无法复现（原本概率极低）。

需要加大检查范围，查看 bsp log：

```

<...>-1039 [007] .... 45605.235726: sched_fork_time: comm=system_server pid=1039 child_comm=system_server child_pid=21526 fork_time=14294154 us
<...>-1039 [007] d..1 45605.238933: signal_generate: sig=11 errno=0 code=196609 comm=system_server pid=1039 grp=0 res=0
<...>-21526 [006] .... 45605.238958: sched_process_exit: comm=Thread=9957 pid=21526 prio=120
<...>-1039 [007] d..1 45605.238965: signal_deliver: sig=11 errno=0 code=196609 sa_handler=5a01a10bb0 sa_flags=18000004

```

发现子线程果然自己退出去了，那么问题来了，还没 unlock，子线程如何偷跑？是否是 lock 失效了？在仔细看汇编代码：

309	AA15A3FA	mov	x0, x21	; x0, thread
NUX:000000777884D300	9400017C	b1	0x777884D8F4	; __pthread_internal_add
NUX:000000777884D304	F90002D0	str	x0, [x20]	
73	885F7EC8	ldxr	u8, [x22]	
NUX:000000777884D30C	8809FEDF	stlxr	u9, u2R, [x22]	
NUX:000000777884D310	35FFFFC9	cbnz	u9, 0x777884D30C	
NUX:000000777884D314	7100091F	cmp	u8, #0x2	; u8, 02
NUX:000000777884D318	540006E1	b.ne	0x777884D3F8	
74	89421280	ldrb	u8, [x21, #0xD4]	; u8, [thread, #132]
NUX:000000777884D320	48			
4B	52000C48	mov	u0, #0x02	; u0, 02
NUX:000000777884D324				

对应代码：

```

72 void unlock() {
73     if (atomic_exchange_explicit(&state, Unlocked, memory_order_release) == LockedWithWaiter) {
74         __futex_wake_ex(&state, process_shared, 1); /* crash here */
75     }
76 }

```

一看，问题就明了了，在 73 行的时候已经 unlock 了，如果子线程被意外唤醒，那么是有可能先跑然后调用 pthread_exit() 把 thread 释放，导致了 lock 也一起释放，那么在 74 行的 process_shared 就无法访问，访问就会发生 NE。

解决方法

google patch: [https://android-review.googlesource.com/c/platform/bionic/+/937824](https://android-review.googlesource.com/c/platform/bionic/+/)

结语

深入汇编，仔细看现场。

利用 bsp log 观察流程。

附录

各种标准

SUS - (Single UNIX® Specification)
<http://www.unix.org/online.html> (注册一个账号即可下载)

LSB - (Linux Standard Base: linux 标准规范)
<http://refspecs.linuxbase.org/lsb.shtml>

DWARF - (调试信息格式)

<http://dwarfstd.org/>

gABI - (System V Application Binary Interface)

<http://refspecs.linuxbase.org/elf/gabi41.pdf>

C++ ABI for Itanium

<http://refspecs.linuxbase.org/cxxabi-1.83.html>

ELF - (Executable and Linkable Format)

<http://refspecs.linuxbase.org/elf/elf.pdf>