**一、概述**

Android系统有监控程序异常退出的机制，这便是本文要讲述得debuggerd守护进程。当发生native crash或者主动调用debuggerd时，会输出进程相关的状态信息到文件或者控制台。输出的debuggerd数据 保存在文件/data/tombstones/tombstone\_XX，该类型文件个数上限位10个，当超过时则每次覆盖时间最老的文件。

针对进程出现的不同的状态，Linux kernel会发送相应的signal给异常进程，捕获signal并对其做相应的处理（通常动作是退出异常进程）。而Android在这机制的前提下，通过拦截这些信号来dump进程信息，方便开发人员调试分析。

debuggerd守护进程会打开socket服务端，当需要调用debuggerd服务时，先通过客户端进程向debuggerd服务端建立socket连接，然后发送不同的请求给debuggerd服务端，当服务端收到不同的请求，则会采取相应的dump操作。

接下来从源码角度来探索debuggerd客户端和服务端的工作原理。

**二、debuggerd客户端**

debuggerd -b <tid>

debuggerd <tid>

通过adb执行上面的命令都能触发debuggerd进行相应的dump操作，其中参数-b表示在控制台中输出backtrace，参数tid表示的是需要dump的进程或者线程id。这两个命令的输出结果相差较大，下面来一步步分析看看这两个命令分别能触发哪些操作，执行上述命令都会调用debuggerd的main方法()。

**2.1 main**

[-> /debuggerd/debuggerd.cpp]

int main(int argc, char\*\* argv) {

...

bool dump\_backtrace = false;

bool have\_tid = false;

pid\_t tid = 0;

//参数解析backtrace与tid信息

for (int i = 1; i < argc; i++) {

if (!strcmp(argv[i], "-b")) {

dump\_backtrace = true;

} else if (!have\_tid) {

tid = atoi(argv[i]);

have\_tid = true;

} else {

usage();

return 1;

}

}

//没有指定tid则直接返回

if (!have\_tid) {

usage();

return 1;

}

//【见小节2.2】

return do\_explicit\_dump(tid, dump\_backtrace);

}

对于debuggerd命令，必须指定线程tid，否则不做任何操作，直接返回。

**2.2 do\_explicit\_dump**

[-> /debuggerd/debuggerd.cpp]

static int do\_explicit\_dump(pid\_t tid, bool dump\_backtrace) {

fprintf(stdout, "Sending request to dump task %d.\n", tid);

if (dump\_backtrace) {

fflush(stdout);

//输出到控制台【见小节2.3】

if (dump\_backtrace\_to\_file(tid, fileno(stdout)) < 0) {

fputs("Error dumping backtrace.\n", stderr);

return 1;

}

} else {

char tombstone\_path[PATH\_MAX];

//输出到tombstone文件【见小节2.4】

if (dump\_tombstone(tid, tombstone\_path, sizeof(tombstone\_path)) < 0) {

fputs("Error dumping tombstone.\n", stderr);

return 1;

}

fprintf(stderr, "Tombstone written to: %s\n", tombstone\_path);

}

return 0;

}

dump\_backtrace等于true代表的是输出backtrace到控制台，否则意味着输出到tombstone文件。

**2.3 dump\_backtrace\_to\_file**

[-> libcutils/debugger.c]

int dump\_backtrace\_to\_file(pid\_t tid, int fd) {

return dump\_backtrace\_to\_file\_timeout(tid, fd, 0);

}

int dump\_backtrace\_to\_file\_timeout(pid\_t tid, int fd, int timeout\_secs) {

//向socket服务端发送dump backtrace的请求【见小节2.5】

int sock\_fd = make\_dump\_request(DEBUGGER\_ACTION\_DUMP\_BACKTRACE, tid, timeout\_secs);

if (sock\_fd < 0) {

return -1;

}

int result = 0;

char buffer[1024];

ssize\_t n;

//阻塞等待，从sock\_fd中读取到服务端发送过来的数据，并写入buffer

while ((n = TEMP\_FAILURE\_RETRY(read(sock\_fd, buffer, sizeof(buffer)))) > 0) {

//再将buffer数据输出到fd，此处是stdout文件描述符(屏幕终端)。

if (TEMP\_FAILURE\_RETRY(write(fd, buffer, n)) != n) {

result = -1;

break;

}

}

close(sock\_fd);

return result;

}

该方法的功能：

* 首先，向debuggerd的socket服务端发出DEBUGGER\_ACTION\_DUMP\_BACKTRACE请求，然后阻塞等待；
* 循环遍历读取debuggerd服务端发送过来的数据，并写入到buffer;
* 再将buffer数据输出到fd，此处是stdout文件描述符(屏幕终端)。

**2.4 dump\_tombstone**

[-> libcutils/debugger.c]

int dump\_tombstone(pid\_t tid, char\* pathbuf, size\_t pathlen) {

return dump\_tombstone\_timeout(tid, pathbuf, pathlen, 0);

}

int dump\_tombstone\_timeout(pid\_t tid, char\* pathbuf, size\_t pathlen, int timeout\_secs) {

//向socket服务端发送dump tombstone的请求【见小节2.5】

int sock\_fd = make\_dump\_request(DEBUGGER\_ACTION\_DUMP\_TOMBSTONE, tid, timeout\_secs);

if (sock\_fd < 0) {

return -1;

}

char buffer[100];

int result = 0; ，

//从sock\_fd中读取到服务端发送过来的tombstone文件名，并写入buffer

ssize\_t n = TEMP\_FAILURE\_RETRY(read(sock\_fd, buffer, sizeof(buffer) - 1));

if (n <= 0) {

result = -1;

} else {

if (pathbuf && pathlen) {

if (n >= (ssize\_t) pathlen) {

n = pathlen - 1;

}

buffer[n] = '\0';

//将buffer数据拷贝到pathbuf

memcpy(pathbuf, buffer, n + 1);

}

}

close(sock\_fd);

return result;

}

该方法的功能：

* 首先，向debuggerd的socket服务端发出DEBUGGER\_ACTION\_DUMP\_TOMBSTONE请求，然后阻塞等待；
* 循环遍历读取debuggerd服务端发送过来的tombstone文件名，并写入到buffer;
* 将buffer数据拷贝到pathbuf，即拷贝tombstone文件名。

**2.5 make\_dump\_request**

[-> libcutils/debugger.c]

static int make\_dump\_request(debugger\_action\_t action, pid\_t tid, int timeout\_secs) {

debugger\_msg\_t msg;

memset(&msg, 0, sizeof(msg));

msg.tid = tid;

msg.action = action;

//与debuggerd服务端建立socket通信，获取client端描述符sock\_fd

int sock\_fd = socket\_local\_client(DEBUGGER\_SOCKET\_NAME, ANDROID\_SOCKET\_NAMESPACE\_ABSTRACT,

SOCK\_STREAM | SOCK\_CLOEXEC);

...

//通过write()方法将msg信息写入文件描述符sock\_fd【见小节2.6】

if (send\_request(sock\_fd, &msg, sizeof(msg)) < 0) {

close(sock\_fd);

return -1;

}

return sock\_fd;

}

该函数的功能是与debuggerd服务端建立socket通信，并发送action请求，以执行相应操作。

**2.6 send\_request**

static int send\_request(int sock\_fd, void\* msg\_ptr, size\_t msg\_len) {

int result = 0;

//写入消息

if (TEMP\_FAILURE\_RETRY(write(sock\_fd, msg\_ptr, msg\_len)) != (ssize\_t) msg\_len) {

result = -1;

} else {

char ack;

//等待应答消息

if (TEMP\_FAILURE\_RETRY(read(sock\_fd, &ack, 1)) != 1) {

result = -1;

}

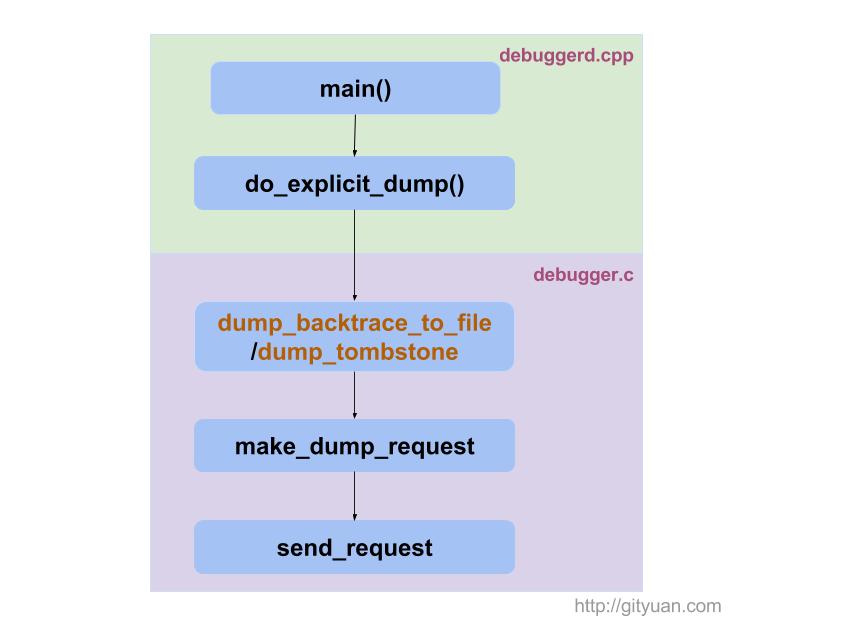
}

return result;

}

**2.7 小节**

通过调用debuggerd <tid>命令调用流程图：



执行debuggerd命令最终都是调用send\_request()方法，向debuggerd服务端发出DEBUGGER\_ACTION\_DUMP\_TOMBSTONE或者DEBUGGER\_ACTION\_DUMP\_BACKTRACE请求，那对于debuggerd服务端收到相应命令做了哪些操作呢，要想明白这个过程，接下来看看debuggerd服务端的工作。

## debuggerd服务端

在执行debuggerd命令之前，debuggerd服务端早早就以准备就绪，时刻等待着client请求的到来。

### 3.1 debuggerd.rc

由init进程fork子进程来以daemon方式启动，定义在debuggerd.rc文件（旧版本位于init.rc）

service debuggerd /system/bin/debuggerd

group root readproc

writepid /dev/cpuset/system-background/tasks

init进程会解析上述rc文件，调用/system/bin/debuggerd文件，进入main方法，此时不带有任何参数。 接下来进入main()方法。

### 3.2 main

[-> /debuggerd/debuggerd.cpp]

int main(int argc, char\*\* argv) {

union selinux\_callback cb;

//当参数个数为1则启动服务

if (argc == 1) {

cb.func\_audit = audit\_callback;

selinux\_set\_callback(SELINUX\_CB\_AUDIT, cb);

cb.func\_log = selinux\_log\_callback;

selinux\_set\_callback(SELINUX\_CB\_LOG, cb);

//【见小节3.3】

return do\_server();

}

...

}

### 3.3 do\_server

[-> /debuggerd/debuggerd.cpp]

static int do\_server() {

//忽略debuggerd进程自身crash的处理过程。重置所有crash handlers

signal(SIGABRT, SIG\_DFL);

signal(SIGBUS, SIG\_DFL);

signal(SIGFPE, SIG\_DFL);

signal(SIGILL, SIG\_DFL);

signal(SIGSEGV, SIG\_DFL);

#ifdef SIGSTKFLT

signal(SIGSTKFLT, SIG\_DFL);

#endif

signal(SIGTRAP, SIG\_DFL);

//忽略向已关闭socket执行写操作失败的信号

signal(SIGPIPE, SIG\_IGN);

//阻塞SIGCHLD

sigset\_t sigchld;

sigemptyset(&sigchld);

sigaddset(&sigchld, SIGCHLD);

sigprocmask(SIG\_SETMASK, &sigchld, nullptr);

//建立socket通信中的服务端

int s = socket\_local\_server(SOCKET\_NAME, ANDROID\_SOCKET\_NAMESPACE\_ABSTRACT,

SOCK\_STREAM | SOCK\_CLOEXEC);

if (s == -1) return 1;

// Fork子进程来发送信号(同样具有root权限)，并监听pipe来暂停和恢复目标进程

if (!start\_signal\_sender()) {

ALOGE("debuggerd: failed to fork signal sender");

return 1;

}

ALOGI("debuggerd: starting\n");

for (;;) {

sockaddr\_storage ss;

sockaddr\* addrp = reinterpret\_cast<sockaddr\*>(&ss);

socklen\_t alen = sizeof(ss);

//等待客户端连接

ALOGV("waiting for connection\n");

int fd = accept4(s, addrp, &alen, SOCK\_CLOEXEC);

if (fd == -1) {

ALOGE("accept failed: %s\n", strerror(errno));

continue;

}

//处理新连接的客户端请求【见小节3.4】

handle\_request(fd);

}

return 0;

}

主要功能：

* 忽略debuggerd进程自身crash的处理过程；
* 重置所有crash handlers；
* 建立socket通信中的服务端；
* 循环等待客户端的连接，并调用handle\_request处理客户端请求。

### 3.4 handle\_request

[-> /debuggerd/debuggerd.cpp]

static void handle\_request(int fd) {

ALOGV("handle\_request(%d)\n", fd);

android::base::unique\_fd closer(fd);

debugger\_request\_t request;

memset(&request, 0, sizeof(request));

//读取client发送过来的请求【见小节3.5】

int status = read\_request(fd, &request);

if (status != 0) {

return;

}

#if defined(\_\_LP64\_\_)

//对于32位的进程，重定向到32位debuggerd

if (is32bit(request.tid)) {

//仅仅dump backtrace和tombstone请求能重定向

if (request.action == DEBUGGER\_ACTION\_DUMP\_BACKTRACE ||

request.action == DEBUGGER\_ACTION\_DUMP\_TOMBSTONE) {

redirect\_to\_32(fd, &request);

}

return;

}

#endif

//fork子进程来处理其余请求命令

pid\_t fork\_pid = fork();

if (fork\_pid == -1) {

ALOGE("debuggerd: failed to fork: %s\n", strerror(errno));

} else if (fork\_pid == 0) {

//子进程执行【见小节3.6】

worker\_process(fd, request);

} else {

//父进程执行【见小节3.7】

monitor\_worker\_process(fork\_pid, request);

}

}

### 3.5 read\_request

[-> /debuggerd/debuggerd.cpp]

static int read\_request(int fd, debugger\_request\_t\* out\_request) {

ucred cr;

socklen\_t len = sizeof(cr);

//从fd获取client进程的pid,uid,gid

int status = getsockopt(fd, SOL\_SOCKET, SO\_PEERCRED, &cr, &len);

...

fcntl(fd, F\_SETFL, O\_NONBLOCK);

pollfd pollfds[1];

pollfds[0].fd = fd;

pollfds[0].events = POLLIN;

pollfds[0].revents = 0;

//读取tid

status = TEMP\_FAILURE\_RETRY(poll(pollfds, 1, 3000));

debugger\_msg\_t msg;

memset(&msg, 0, sizeof(msg));

//从fd读取数据并保存到结构体msg

status = TEMP\_FAILURE\_RETRY(read(fd, &msg, sizeof(msg)));

...

out\_request->action = static\_cast<debugger\_action\_t>(msg.action);

out\_request->tid = msg.tid;

out\_request->pid = cr.pid;

out\_request->uid = cr.uid;

out\_request->gid = cr.gid;

out\_request->abort\_msg\_address = msg.abort\_msg\_address;

out\_request->original\_si\_code = msg.original\_si\_code;

if (msg.action == DEBUGGER\_ACTION\_CRASH) {

// C/C++进程crash时发送过来的请求

char buf[64];

struct stat s;

snprintf(buf, sizeof buf, "/proc/%d/task/%d", out\_request->pid, out\_request->tid);

if (stat(buf, &s)) {

return -1; //tid不存在，忽略该显式dump请求

}

} else if (cr.uid == 0

|| (cr.uid == AID\_SYSTEM && msg.action == DEBUGGER\_ACTION\_DUMP\_BACKTRACE)) {

//root权限既可以可收集backtraces，又可以dump tombstones；

//system权限只允许收集backtraces;

status = get\_process\_info(out\_request->tid, &out\_request->pid,

&out\_request->uid, &out\_request->gid);

if (status < 0) {

return -1; //tid不存在，忽略该显式dump请求

}

if (!selinux\_action\_allowed(fd, out\_request))

return -1; //selinux权限不足，忽略该请求

} else {

//其他情况，则直接忽略

return -1;

}

return 0;

}

该方法的功能是首先从socket获取client进程的pid,uid,gid用于权限控制，能处理以下三种情况：

* C/C++进程crash时发送过来的请求；
* root权限既可以可收集backtraces，又可以dump tombstones；
* system权限只允许收集backtraces。

针对这些情况若相应的tid不存在或selinux权限不满足，则都忽略该显式dump请求。read\_request执行完成后，则从socket通道中读取到request信息。

### 3.6 worker\_process

处于client发送过来的请求，server端通过子进程来处理

[-> /debuggerd/debuggerd.cpp]

static void worker\_process(int fd, debugger\_request\_t& request) {

std::string tombstone\_path;

int tombstone\_fd = -1;

switch (request.action) {

case DEBUGGER\_ACTION\_DUMP\_TOMBSTONE: //case1:输出tombstone文件

case DEBUGGER\_ACTION\_CRASH: //case2:出现native crash

//打开tombstone文件【见小节3.6.1】

tombstone\_fd = open\_tombstone(&tombstone\_path);

if (tombstone\_fd == -1) {

exit(1); //无法打开tombstone文件，则退出该进程

}

break;

case DEBUGGER\_ACTION\_DUMP\_BACKTRACE: //case3:输出backtrace

break;

default:

exit(1); //其他case则直接结束进程

}

// Attach到目标进程

if (ptrace(PTRACE\_ATTACH, request.tid, 0, 0) != 0) {

exit(1); //attach失败则退出该进程

}

bool attach\_gdb = should\_attach\_gdb(request);

if (attach\_gdb) {

// 在特权模式降级之前，打开所有需要监听的input设备

if (init\_getevent() != 0) {

attach\_gdb = false; //初始化input设备失败，不再等待gdb

}

}

...

//生成backtrace【见小节3.6.2】

std::unique\_ptr<BacktraceMap> backtrace\_map(BacktraceMap::Create(request.pid));

int amfd = -1;

std::unique\_ptr<std::string> amfd\_data;

if (request.action == DEBUGGER\_ACTION\_CRASH) {

//当发生native crash，则连接到AMS【见小节3.6.3】

amfd = activity\_manager\_connect();

amfd\_data.reset(new std::string);

}

bool succeeded = false;

//取消特权模式

if (!drop\_privileges()) {

\_exit(1); //操作失败，则退出

}

int crash\_signal = SIGKILL;

//执行dump操作，【见小节3.6.4】

succeeded = perform\_dump(request, fd, tombstone\_fd, backtrace\_map.get(), siblings,

&crash\_signal, amfd\_data.get());

if (succeeded) {

if (request.action == DEBUGGER\_ACTION\_DUMP\_TOMBSTONE) {

if (!tombstone\_path.empty()) {

android::base::WriteFully(fd, tombstone\_path.c\_str(), tombstone\_path.length());

}

}

}

if (attach\_gdb) {

//向目标进程发送SIGSTOP信号

if (!send\_signal(request.pid, 0, SIGSTOP)) {

attach\_gdb = false; //无法停止通过gdb attach的进程

}

}

if (!attach\_gdb) {

//将进程crash情况告知AMS【见小节3.6.5】

activity\_manager\_write(request.pid, crash\_signal, amfd, \*amfd\_data.get());

}

//detach目标进程

ptrace(PTRACE\_DETACH, request.tid, 0, 0);

for (pid\_t sibling : siblings) {

ptrace(PTRACE\_DETACH, sibling, 0, 0);

}

if (!attach\_gdb && request.action == DEBUGGER\_ACTION\_CRASH) {

//发送信号SIGKILL给目标进程

if (!send\_signal(request.pid, request.tid, crash\_signal)) {

ALOGE("debuggerd: failed to kill process %d: %s", request.pid, strerror(errno));

}

}

//如果需要则等待gdb

if (attach\_gdb) {

wait\_for\_user\_action(request);

//将进程crash情况告知AMS

activity\_manager\_write(request.pid, crash\_signal, amfd, \*amfd\_data.get());

//发送信号SIGCONT给目标进程

if (!send\_signal(request.pid, 0, SIGCONT)) {

ALOGE("debuggerd: failed to resume process %d: %s", request.pid, strerror(errno));

}

uninit\_getevent();

}

close(amfd);

exit(!succeeded);

}

这个流程比较长，这里介绍attach\_gdb=false的执行流程

#### 3.6.1 open\_tombstone

[-> tombstone.cpp]

int open\_tombstone(std::string\* out\_path) {

char path[128];

int fd = -1;

int oldest = -1;

struct stat oldest\_sb;

//遍历查找

for (int i = 0; i < MAX\_TOMBSTONES; i++) {

snprintf(path, sizeof(path), TOMBSTONE\_TEMPLATE, i);

struct stat sb;

if (stat(path, &sb) == 0) {

//记录修改时间最老的tombstone文件

if (oldest < 0 || sb.st\_mtime < oldest\_sb.st\_mtime) {

oldest = i;

oldest\_sb.st\_mtime = sb.st\_mtime;

}

continue;

}

//存在没有使用的tombstone文件，则打开并赋给out\_path，然后直接返回

fd = open(path, O\_CREAT | O\_EXCL | O\_WRONLY | O\_NOFOLLOW | O\_CLOEXEC, 0600);

if (out\_path) {

\*out\_path = path;

}

fchown(fd, AID\_SYSTEM, AID\_SYSTEM);

return fd;

}

//找不到最老的可用tombstone文件，则默认使用tombstone 0

if (oldest < 0) {

oldest = 0;

}

snprintf(path, sizeof(path), TOMBSTONE\_TEMPLATE, oldest);

//打开最老的tombstone文件

fd = open(path, O\_CREAT | O\_TRUNC | O\_WRONLY | O\_NOFOLLOW | O\_CLOEXEC, 0600);

...

if (out\_path) {

\*out\_path = path;

}

fchown(fd, AID\_SYSTEM, AID\_SYSTEM);

return fd;

}

其中TOMBSTONE\_TEMPLATE为data/tombstones/tombstone\_%02d，文件个数上限MAX\_TOMBSTONES=10

打开tombstone文件规则：

1. 当已使用的tombstone文件个数小于10时，则创建新的tombstone文件；否则执行2；
2. 获取修改时间最老的tombstone文件，并覆写该文件；
3. 如果最老文件不存在，则默认使用文件data/tombstones/tombstone\_00

#### 3.6.2 BacktraceMap::Create

[-> BacktraceMap.cpp]

BacktraceMap\* BacktraceMap::Create(pid\_t pid, bool /\*uncached\*/) {

BacktraceMap\* map = new BacktraceMap(pid);

if (!map->Build()) {

delete map;

return nullptr;

}

return map;

}

解析/proc/[pid]/maps, 生成BacktraceMap.

#### 3.6.3 activity\_manager\_connect

[-> debuggerd.cpp]

static int activity\_manager\_connect() {

android::base::unique\_fd amfd(socket(PF\_UNIX, SOCK\_STREAM, 0));

if (amfd.get() < -1) {

return -1; ///无法连接到ActivityManager(socket失败)

}

struct sockaddr\_un address;

memset(&address, 0, sizeof(address));

address.sun\_family = AF\_UNIX;

//该路径必须匹配NativeCrashListener.java中的定义

strncpy(address.sun\_path, "/data/system/ndebugsocket", sizeof(address.sun\_path));

if (TEMP\_FAILURE\_RETRY(connect(amfd.get(), reinterpret\_cast<struct sockaddr\*>(&address),

sizeof(address))) == -1) {

return -1; //无法连接到ActivityManager(connect失败)

}

struct timeval tv;

memset(&tv, 0, sizeof(tv));

tv.tv\_sec = 1;

if (setsockopt(amfd.get(), SOL\_SOCKET, SO\_SNDTIMEO, &tv, sizeof(tv)) == -1) {

ALOGE("debuggerd: Unable to connect to activity manager (setsockopt SO\_SNDTIMEO failed: %s)",

strerror(errno));

return -1; //无法连接到ActivityManager(setsockopt SO\_SNDTIMEO失败)

}

tv.tv\_sec = 3;

if (setsockopt(amfd.get(), SOL\_SOCKET, SO\_RCVTIMEO, &tv, sizeof(tv)) == -1) {

ALOGE("debuggerd: Unable to connect to activity manager (setsockopt SO\_RCVTIMEO failed: %s)",

strerror(errno));

return -1; //无法连接到ActivityManager(setsockopt SO\_RCVTIMEO失败)

}

return amfd.release();

}

该方法的功能是建立与ActivityManager的socket连接。

#### 3.6.4 perform\_dump

根据接收到不同的signal采取相应的操作

[-> debuggerd.cpp]

static bool perform\_dump(const debugger\_request\_t& request, int fd, int tombstone\_fd, BacktraceMap\* backtrace\_map, const std::set<pid\_t>& siblings, int\* crash\_signal, std::string\* amfd\_data) {

if (TEMP\_FAILURE\_RETRY(write(fd, "\0", 1)) != 1) {

ALOGE("debuggerd: failed to respond to client: %s\n", strerror(errno));

return false; //无法响应client端请求

}

int total\_sleep\_time\_usec = 0;

while (true) {

//等待信号到来

int signal = wait\_for\_signal(request.tid, &total\_sleep\_time\_usec);

switch (signal) {

case -1:

ALOGE("debuggerd: timed out waiting for signal");

return false; //等待超时

case SIGSTOP:

if (request.action == DEBUGGER\_ACTION\_DUMP\_TOMBSTONE) {

ALOGV("debuggerd: stopped -- dumping to tombstone");

//【见小节4.1】

engrave\_tombstone(tombstone\_fd, backtrace\_map, request.pid, request.tid, siblings, signal,

request.original\_si\_code, request.abort\_msg\_address, amfd\_data);

} else if (request.action == DEBUGGER\_ACTION\_DUMP\_BACKTRACE) {

ALOGV("debuggerd: stopped -- dumping to fd");

//【见小节4.4】

dump\_backtrace(fd, backtrace\_map, request.pid, request.tid, siblings, nullptr);

} else {

ALOGV("debuggerd: stopped -- continuing");

if (ptrace(PTRACE\_CONT, request.tid, 0, 0) != 0) {

ALOGE("debuggerd: ptrace continue failed: %s", strerror(errno));

return false;

}

continue; //再次循环

}

break;

case SIGABRT:

case SIGBUS:

case SIGFPE:

case SIGILL:

case SIGSEGV:

#ifdef SIGSTKFLT

case SIGSTKFLT:

#endif

case SIGTRAP:

ALOGV("stopped -- fatal signal\n");

\*crash\_signal = signal;

//【见小节4.1】

engrave\_tombstone(tombstone\_fd, backtrace\_map, request.pid, request.tid, siblings, signal,

request.original\_si\_code, request.abort\_msg\_address, amfd\_data);

break;

default:

ALOGE("debuggerd: process stopped due to unexpected signal %d\n", signal);

break;

}

break;

}

return true;

}

致命信号有SIGABRT，SIGBUS，SIGFPE，SIGILL，SIGSEGV，SIGSTKFLT，SIGTRAP共7个信息，能造成native crash。

#### 3.6.5 activity\_manager\_write

[-> debuggerd.cpp]

static void activity\_manager\_write(int pid, int signal, int amfd, const std::string& amfd\_data) {

if (amfd == -1) {

return;

}

//写入32-bit pid和signal，以及原始dump信息，最后添加0以标记结束

uint32\_t datum = htonl(pid);

if (!android::base::WriteFully(amfd, &datum, 4)) {

return; //AM pid写入失败

}

datum = htonl(signal);

if (!android::base::WriteFully(amfd, &datum, 4)) {

return;//AM signal写入失败

}

if (!android::base::WriteFully(amfd, amfd\_data.c\_str(), amfd\_data.size())) {

return;//AM data写入失败

}

uint8\_t eodMarker = 0;

if (!android::base::WriteFully(amfd, &eodMarker, 1)) {

return; //AM eod 写入失败

}

//读取应答消息，如果3s超时未收到则读取失败

android::base::ReadFully(amfd, &eodMarker, 1);

}

### 3.7 monitor\_worker\_process

父进程处理

[-> debuggerd.cpp]

static void monitor\_worker\_process(int child\_pid, const debugger\_request\_t& request) {

struct timespec timeout = {.tv\_sec = 10, .tv\_nsec = 0 };

if (should\_attach\_gdb(request)) {

timeout.tv\_sec = INT\_MAX;

}

sigset\_t signal\_set;

sigemptyset(&signal\_set);

sigaddset(&signal\_set, SIGCHLD);

bool kill\_worker = false;

bool kill\_target = false;

bool kill\_self = false;

int status;

siginfo\_t siginfo;

int signal = TEMP\_FAILURE\_RETRY(sigtimedwait(&signal\_set, &siginfo, &timeout));

if (signal == SIGCHLD) {

pid\_t rc = waitpid(-1, &status, WNOHANG | WUNTRACED);

if (rc != child\_pid) {

ALOGE("debuggerd: waitpid returned unexpected pid (%d), committing murder-suicide", rc);

if (WIFEXITED(status)) {

ALOGW("debuggerd: pid %d exited with status %d", rc, WEXITSTATUS(status));

} else if (WIFSIGNALED(status)) {

ALOGW("debuggerd: pid %d received signal %d", rc, WTERMSIG(status));

} else if (WIFSTOPPED(status)) {

ALOGW("debuggerd: pid %d stopped by signal %d", rc, WSTOPSIG(status));

} else if (WIFCONTINUED(status)) {

ALOGW("debuggerd: pid %d continued", rc);

}

kill\_worker = true;

kill\_target = true;

kill\_self = true;

} else if (WIFSIGNALED(status)) {

ALOGE("debuggerd: worker process %d terminated due to signal %d", child\_pid, WTERMSIG(status));

kill\_worker = false;

kill\_target = true;

} else if (WIFSTOPPED(status)) {

ALOGE("debuggerd: worker process %d stopped due to signal %d", child\_pid, WSTOPSIG(status));

kill\_worker = true;

kill\_target = true;

}

} else {

ALOGE("debuggerd: worker process %d timed out", child\_pid);

kill\_worker = true;

kill\_target = true;

}

if (kill\_worker) {

// Something bad happened, kill the worker.

if (kill(child\_pid, SIGKILL) != 0) {

ALOGE("debuggerd: failed to kill worker process %d: %s", child\_pid, strerror(errno));

} else {

waitpid(child\_pid, &status, 0);

}

}

int exit\_signal = SIGCONT;

if (kill\_target && request.action == DEBUGGER\_ACTION\_CRASH) {

ALOGE("debuggerd: killing target %d", request.pid);

exit\_signal = SIGKILL;

} else {

ALOGW("debuggerd: resuming target %d", request.pid);

}

if (kill(request.pid, exit\_signal) != 0) {

ALOGE("debuggerd: failed to send signal %d to target: %s", exit\_signal, strerror(errno));

}

if (kill\_self) {

stop\_signal\_sender();

\_exit(1);

}

}

### 3.8 小节

调用流程：

debuggerd.main

do\_server

handle\_request

read\_request

worker\_process(子进程)

monitor\_worker\_process(父进程)

整个过程的核心方法为worker\_process()，其流程如下：

1. 根据请worker\_process()求中的不同action采取相应操作，除此之外则立即结束进程
   * DEBUGGER\_ACTION\_DUMP\_TOMBSTONE，则调用open\_tombstone并继续执行；
   * DEBUGGER\_ACTION\_CRASH ，则调用open\_tombstone并继续执行；
   * DEBUGGER\_ACTION\_DUMP\_BACKTRACE，则直接继续执行；
2. 调用ptrace方法attach到目标进程;
3. 调用BacktraceMap::Create来生成backtrace;
4. 当Action=DEBUGGER\_ACTION\_CRASH，则执行activity\_manager\_connect；
5. 调用drop\_privileges来取消特权模式；
6. 通过perform\_dump执行dump操作：【见小节3.6.4】
   * SIGSTOP && DEBUGGER\_ACTION\_DUMP\_BACKTRACE，则dump\_backtrace()
   * SIGSTOP && DEBUGGER\_ACTION\_DUMP\_TOMBSTONE，则engrave\_tombstone()
   * SIGBUS等致命信号，则engrave\_tombstone()
7. 当Action=DEBUGGER\_ACTION\_DUMP\_TOMBSTONE，则将向client端写入tombstone数据；
8. 调用activity\_manager\_write，将进程crash情况告知AMS；
9. 调用ptrace方法detach到目标进程;
10. 当Action=DEBUGGER\_ACTION\_CRASH，发送信号SIGKILL给目标进程tid
11. 调用exit来结束进程。

整个过程中，【见小节3.6.4】perform\_dump是核心过程：对于DEBUGGER\_ACTION\_DUMP\_BACKTRACE命令，则执行dump\_backtrace；否则执行engrave\_tombstone。接下来分别说说这两个过程

## tombstone

### 4.1 engrave\_tombstone

[-> debuggerd/tombstone.cpp]

void engrave\_tombstone(int tombstone\_fd, BacktraceMap\* map, pid\_t pid, pid\_t tid, const std::set<pid\_t>& siblings, int signal, int original\_si\_code, uintptr\_t abort\_msg\_address, std::string\* amfd\_data) {

log\_t log;

log.current\_tid = tid;

log.crashed\_tid = tid;

if (tombstone\_fd < 0) {

ALOGE("debuggerd: skipping tombstone write, nothing to do.\n");

return;

}

log.tfd = tombstone\_fd;

log.amfd\_data = amfd\_data;

//【见小节4.2】

dump\_crash(&log, map, pid, tid, siblings, signal, original\_si\_code, abort\_msg\_address);

}

### 4.2 dump\_crash

[-> debuggerd/tombstone.cpp]

// Dump该pid所对应进程的所有tombstone信息

static void dump\_crash(log\_t\* log, BacktraceMap\* map, pid\_t pid, pid\_t tid, const std::set<pid\_t>& siblings, int signal, int si\_code, uintptr\_t abort\_msg\_address) {

char value[PROPERTY\_VALUE\_MAX];

//当ro.debuggable =1，则输出log信息

property\_get("ro.debuggable", value, "0");

bool want\_logs = (value[0] == '1');

\_LOG(log, logtype::HEADER,

"\*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\*\n");

//tombstone头部信息【见小节4.3】

dump\_header\_info(log);

//tombstone线程信息【见小节4.4】

dump\_thread(log, pid, tid, map, signal, si\_code, abort\_msg\_address, true);

if (want\_logs) {

//输出log信息【见小节4.5】

dump\_logs(log, pid, 5);

}

if (!siblings.empty()) {

for (pid\_t sibling : siblings) {

//tombstone兄弟线程信息【见小节4.4】

dump\_thread(log, pid, sibling, map, 0, 0, 0, false);

}

}

if (want\_logs) {

dump\_logs(log, pid, 0);

}

}

主要输出信息：

* dump\_header\_info
* 主线程dump\_thread
* dump\_logs (ro.debuggable=1 才输出此项)
* 兄弟线程dump\_thread
* dump\_logs (ro.debuggable=1 才输出此项)

### 4.3 dump\_header\_info

[-> debuggerd/tombstone.cpp]

static void dump\_header\_info(log\_t\* log) {

char fingerprint[PROPERTY\_VALUE\_MAX];

char revision[PROPERTY\_VALUE\_MAX];

property\_get("ro.build.fingerprint", fingerprint, "unknown");

property\_get("ro.revision", revision, "unknown");

\_LOG(log, logtype::HEADER, "Build fingerprint: '%s'\n", fingerprint);

\_LOG(log, logtype::HEADER, "Revision: '%s'\n", revision);

\_LOG(log, logtype::HEADER, "ABI: '%s'\n", ABI\_STRING);

}

例如：

Build fingerprint: 'xxx/xxx/MMB29M/gityuan06080845:userdebug/test-keys'

Revision: '0'

ABI: 'arm'

### 4.4 dump\_thread(主)

调用方法dump\_thread(log, pid, tid, map, signal, si\_code, abort\_msg\_address, true);

[-> debuggerd/tombstone.cpp]

static void dump\_thread(log\_t\* log, pid\_t pid, pid\_t tid, BacktraceMap\* map, int signal, int si\_code, uintptr\_t abort\_msg\_address, bool primary\_thread) {

log->current\_tid = tid;

//【见小节4.4.1】

dump\_thread\_info(log, pid, tid);

if (signal) {

//【见小节4.4.2】

dump\_signal\_info(log, tid, signal, si\_code);

}

//【见小节3.6.2】

std::unique\_ptr<Backtrace> backtrace(Backtrace::Create(pid, tid, map));

if (primary\_thread) {

//【见小节4.4.3】

dump\_abort\_message(backtrace.get(), log, abort\_msg\_address);

}

//【见小节4.4.4】

dump\_registers(log, tid);

if (backtrace->Unwind(0)) {

//【见小节4.4.5】

dump\_backtrace\_and\_stack(backtrace.get(), log);

} else {

ALOGE("Unwind failed: pid = %d, tid = %d", pid, tid);

}

if (primary\_thread) {

//【见小节4.4.6】

dump\_memory\_and\_code(log, backtrace.get());

if (map) {

//【见小节4.4.7】

dump\_all\_maps(backtrace.get(), map, log, tid);

}

}

log->current\_tid = log->crashed\_tid;

}

#### 4.4.1 dump\_thread\_info

static void dump\_thread\_info(log\_t\* log, pid\_t pid, pid\_t tid) {

char path[64];

char threadnamebuf[1024];

char\* threadname = nullptr;

FILE \*fp;

//获取/proc/<tid>/comm节点的线程名

snprintf(path, sizeof(path), "/proc/%d/comm", tid);

if ((fp = fopen(path, "r"))) {

threadname = fgets(threadnamebuf, sizeof(threadnamebuf), fp);

fclose(fp);

if (threadname) {

size\_t len = strlen(threadname);

if (len && threadname[len - 1] == '\n') {

threadname[len - 1] = '\0';

}

}

}

// Blacklist logd, logd.reader, logd.writer, logd.auditd, logd.control ...

static const char logd[] = "logd";

if (threadname != nullptr && !strncmp(threadname, logd, sizeof(logd) - 1)

&& (!threadname[sizeof(logd) - 1] || (threadname[sizeof(logd) - 1] == '.'))) {

log->should\_retrieve\_logcat = false;

}

char procnamebuf[1024];

char\* procname = nullptr;

//获取/proc/<pid>/cmdline节点的进程名

snprintf(path, sizeof(path), "/proc/%d/cmdline", pid);

if ((fp = fopen(path, "r"))) {

procname = fgets(procnamebuf, sizeof(procnamebuf), fp);

fclose(fp);

}

\_LOG(log, logtype::HEADER, "pid: %d, tid: %d, name: %s >>> %s <<<\n", pid, tid,

threadname ? threadname : "UNKNOWN", procname ? procname : "UNKNOWN");

}

* 获取进程名：/proc/<pid>/cmdline
* 获取线程名：/proc/<tid>/comm

例如：

//代表system\_server进程的主线程system\_server

pid: 1789, tid: 1789, name: system\_server >>> system\_server <<<

//代表system\_server进程的子线程ActivityManager

pid: 1789, tid: 1827, name: ActivityManager >>> system\_server <<<

#### 4.4.2 dump\_signal\_info

static void dump\_signal\_info(log\_t\* log, pid\_t tid, int signal, int si\_code) {

siginfo\_t si;

memset(&si, 0, sizeof(si));

if (ptrace(PTRACE\_GETSIGINFO, tid, 0, &si) == -1) {

ALOGE("cannot get siginfo: %s\n", strerror(errno));

return;

}

si.si\_code = si\_code;

char addr\_desc[32];

if (signal\_has\_si\_addr(signal)) {

snprintf(addr\_desc, sizeof(addr\_desc), "%p", si.si\_addr);

} else {

snprintf(addr\_desc, sizeof(addr\_desc), "--------");

}

\_LOG(log, logtype::HEADER, "signal %d (%s), code %d (%s), fault addr %s\n",

signal, get\_signame(signal), si.si\_code, get\_sigcode(signal, si.si\_code), addr\_desc);

}

* 对于SIGBUS，SIGFPE，SIGILL，SIGSEGV，SIGTRAP时触发的dump,则会输出fault addr的具体地址，
* 对于SIGSTOP时,则输出fault addr为”——–”

例如：

signal 11 (SIGSEGV), code 1 (SEGV\_MAPERR), fault addr 0xd140109c

signal 19 (SIGSTOP), code 0 (SI\_USER), fault addr --------

此处get\_sigcode函数功能负责根据signal以及si\_code来获取相应信息，下面来列举每种signal所包含的信息种类：

| **signal** | **get\_sigcode** |
| --- | --- |
| SIGILL | ILL\_ILLOPC |
| SIGILL | ILL\_ILLOPN |
| SIGILL | ILL\_ILLADR |
| SIGILL | ILL\_ILLTRP |
| SIGILL | ILL\_PRVOPC |
| SIGILL | ILL\_PRVREG |
| SIGILL | ILL\_COPROC |
| SIGILL | ILL\_BADSTK |
| **signal** | **get\_sigcode** |
| SIGBUS | BUS\_ADRALN |
| SIGBUS | BUS\_ADRERR |
| SIGBUS | BUS\_OBJERR |
| SIGBUS | BUS\_MCEERR\_AR |
| SIGBUS | BUS\_MCEERR\_AO |

| **signal** | **get\_sigcode** |
| --- | --- |
| SIGFPE | FPE\_INTDIV |
| SIGFPE | FPE\_INTOVF |
| SIGFPE | FPE\_FLTDIV |
| SIGFPE | FPE\_FLTOVF |
| SIGFPE | FPE\_FLTUND |
| SIGFPE | FPE\_FLTRES |
| SIGFPE | FPE\_FLTINV |
| SIGFPE | FPE\_FLTSUB |
| **signal** | **get\_sigcode** |
| SIGSEGV | SEGV\_MAPERR |
| SIGSEGV | SEGV\_ACCERR |
| SIGSEGV | SEGV\_BNDERR |
| SIGSEGV | SEGV\_MAPERR |

| **signal** | **get\_sigcode** |
| --- | --- |
| SIGTRAP | TRAP\_BRKPT |
| SIGTRAP | TRAP\_TRACE |
| SIGTRAP | TRAP\_BRANCH |
| SIGTRAP | TRAP\_HWBKPT |

#### 4.4.3 dump\_abort\_message

static void dump\_abort\_message(Backtrace\* backtrace, log\_t\* log, uintptr\_t address) {

if (address == 0) {

return;

}

address += sizeof(size\_t); // Skip the buffer length.

char msg[512];

memset(msg, 0, sizeof(msg));

char\* p = &msg[0];

while (p < &msg[sizeof(msg)]) {

word\_t data;

size\_t len = sizeof(word\_t);

if (!backtrace->ReadWord(address, &data)) {

break;

}

address += sizeof(word\_t);

while (len > 0 && (\*p++ = (data >> (sizeof(word\_t) - len) \* 8) & 0xff) != 0) {

len--;

}

}

msg[sizeof(msg) - 1] = '\0';

\_LOG(log, logtype::HEADER, "Abort message: '%s'\n", msg);

}

#### 4.4.4 dump\_registers

输出系统寄存器信息，这里以arm为例来说明

[-> debuggerd/arm/Machine.cpp]

void dump\_registers(log\_t\* log, pid\_t tid) {

pt\_regs r;

if (ptrace(PTRACE\_GETREGS, tid, 0, &r)) {

ALOGE("cannot get registers: %s\n", strerror(errno));

return;

}

\_LOG(log, logtype::REGISTERS, " r0 %08x r1 %08x r2 %08x r3 %08x\n",

static\_cast<uint32\_t>(r.ARM\_r0), static\_cast<uint32\_t>(r.ARM\_r1),

static\_cast<uint32\_t>(r.ARM\_r2), static\_cast<uint32\_t>(r.ARM\_r3));

\_LOG(log, logtype::REGISTERS, " r4 %08x r5 %08x r6 %08x r7 %08x\n",

static\_cast<uint32\_t>(r.ARM\_r4), static\_cast<uint32\_t>(r.ARM\_r5),

static\_cast<uint32\_t>(r.ARM\_r6), static\_cast<uint32\_t>(r.ARM\_r7));

\_LOG(log, logtype::REGISTERS, " r8 %08x r9 %08x sl %08x fp %08x\n",

static\_cast<uint32\_t>(r.ARM\_r8), static\_cast<uint32\_t>(r.ARM\_r9),

static\_cast<uint32\_t>(r.ARM\_r10), static\_cast<uint32\_t>(r.ARM\_fp));

\_LOG(log, logtype::REGISTERS, " ip %08x sp %08x lr %08x pc %08x cpsr %08x\n",

static\_cast<uint32\_t>(r.ARM\_ip), static\_cast<uint32\_t>(r.ARM\_sp),

static\_cast<uint32\_t>(r.ARM\_lr), static\_cast<uint32\_t>(r.ARM\_pc),

static\_cast<uint32\_t>(r.ARM\_cpsr));

user\_vfp vfp\_regs;

if (ptrace(PTRACE\_GETVFPREGS, tid, 0, &vfp\_regs)) {

ALOGE("cannot get FP registers: %s\n", strerror(errno));

return;

}

for (size\_t i = 0; i < 32; i += 2) {

\_LOG(log, logtype::FP\_REGISTERS, " d%-2d %016llx d%-2d %016llx\n",

i, vfp\_regs.fpregs[i], i+1, vfp\_regs.fpregs[i+1]);

}

\_LOG(log, logtype::FP\_REGISTERS, " scr %08lx\n", vfp\_regs.fpscr);

}

通过ptrace获取寄存器状态信息，这里输出r0-r9,sl,fp,ip,sp,lr,pc,cpsr 以及32个fpregs和一个fpscr.

#### 4.4.5 dump\_backtrace\_and\_stack

[-> debuggerd/tombstone.cpp]

static void dump\_backtrace\_and\_stack(Backtrace\* backtrace, log\_t\* log) {

if (backtrace->NumFrames()) {

\_LOG(log, logtype::BACKTRACE, "\nbacktrace:\n");

//【见小节4.4.5.1】

dump\_backtrace\_to\_log(backtrace, log, " ");

\_LOG(log, logtype::STACK, "\nstack:\n");

//【见小节4.4.5.2】

dump\_stack(backtrace, log);

}

}

**4.4.5.1 输出backtrace信息**

[-> debuggerd/Backtrace.cpp]

void dump\_backtrace\_to\_log(Backtrace\* backtrace, log\_t\* log, const char\* prefix) {

for (size\_t i = 0; i < backtrace->NumFrames(); i++) {

\_LOG(log, logtype::BACKTRACE, "%s%s\n", prefix, backtrace->FormatFrameData(i).c\_str());

}

}

**4.4.5.2 输出stack信息**

[-> debuggerd/tombstone.cpp]

static void dump\_stack(Backtrace\* backtrace, log\_t\* log) {

size\_t first = 0, last;

for (size\_t i = 0; i < backtrace->NumFrames(); i++) {

const backtrace\_frame\_data\_t\* frame = backtrace->GetFrame(i);

if (frame->sp) {

if (!first) {

first = i+1;

}

last = i;

}

}

if (!first) {

return;

}

first--;

// Dump a few words before the first frame.

word\_t sp = backtrace->GetFrame(first)->sp - STACK\_WORDS \* sizeof(word\_t);

dump\_stack\_segment(backtrace, log, &sp, STACK\_WORDS, -1);

// Dump a few words from all successive frames.

// Only log the first 3 frames, put the rest in the tombstone.

for (size\_t i = first; i <= last; i++) {

const backtrace\_frame\_data\_t\* frame = backtrace->GetFrame(i);

if (sp != frame->sp) {

\_LOG(log, logtype::STACK, " ........ ........\n");

sp = frame->sp;

}

if (i == last) {

dump\_stack\_segment(backtrace, log, &sp, STACK\_WORDS, i);

if (sp < frame->sp + frame->stack\_size) {

\_LOG(log, logtype::STACK, " ........ ........\n");

}

} else {

size\_t words = frame->stack\_size / sizeof(word\_t);

if (words == 0) {

words = 1;

} else if (words > STACK\_WORDS) {

words = STACK\_WORDS;

}

dump\_stack\_segment(backtrace, log, &sp, words, i);

}

}

}

#### 4.4.6 dump\_memory\_and\_code

[-> debuggerd/arm/Machine.cpp]

void dump\_memory\_and\_code(log\_t\* log, Backtrace\* backtrace) {

pt\_regs regs;

if (ptrace(PTRACE\_GETREGS, backtrace->Tid(), 0, &regs)) {

ALOGE("cannot get registers: %s\n", strerror(errno));

return;

}

static const char reg\_names[] = "r0r1r2r3r4r5r6r7r8r9slfpipsp";

for (int reg = 0; reg < 14; reg++) {

dump\_memory(log, backtrace, regs.uregs[reg], "memory near %.2s:", &reg\_names[reg \* 2]);

}

dump\_memory(log, backtrace, static\_cast<uintptr\_t>(regs.ARM\_pc), "code around pc:");

if (regs.ARM\_pc != regs.ARM\_lr) {

dump\_memory(log, backtrace, static\_cast<uintptr\_t>(regs.ARM\_lr), "code around lr:");

}

}

#### 4.4.7 dump\_all\_maps

[-> debuggerd/tombstone.cpp]

static void dump\_all\_maps(Backtrace\* backtrace, BacktraceMap\* map, log\_t\* log, pid\_t tid) {

bool print\_fault\_address\_marker = false;

uintptr\_t addr = 0;

siginfo\_t si;

memset(&si, 0, sizeof(si));

if (ptrace(PTRACE\_GETSIGINFO, tid, 0, &si) != -1) {

print\_fault\_address\_marker = signal\_has\_si\_addr(si.si\_signo);

addr = reinterpret\_cast<uintptr\_t>(si.si\_addr);

} else {

ALOGE("Cannot get siginfo for %d: %s\n", tid, strerror(errno));

}

\_LOG(log, logtype::MAPS, "\n");

if (!print\_fault\_address\_marker) {

\_LOG(log, logtype::MAPS, "memory map:\n");

} else {

\_LOG(log, logtype::MAPS, "memory map: (fault address prefixed with --->)\n");

if (map->begin() != map->end() && addr < map->begin()->start) {

\_LOG(log, logtype::MAPS, "--->Fault address falls at %s before any mapped regions\n",

get\_addr\_string(addr).c\_str());

print\_fault\_address\_marker = false;

}

}

std::string line;

for (BacktraceMap::const\_iterator it = map->begin(); it != map->end(); ++it) {

line = " ";

if (print\_fault\_address\_marker) {

if (addr < it->start) {

\_LOG(log, logtype::MAPS, "--->Fault address falls at %s between mapped regions\n",

get\_addr\_string(addr).c\_str());

print\_fault\_address\_marker = false;

} else if (addr >= it->start && addr < it->end) {

line = "--->";

print\_fault\_address\_marker = false;

}

}

line += get\_addr\_string(it->start) + '-' + get\_addr\_string(it->end - 1) + ' ';

if (it->flags & PROT\_READ) {

line += 'r';

} else {

line += '-';

}

if (it->flags & PROT\_WRITE) {

line += 'w';

} else {

line += '-';

}

if (it->flags & PROT\_EXEC) {

line += 'x';

} else {

line += '-';

}

line += android::base::StringPrintf(" %8" PRIxPTR " %8" PRIxPTR,

it->offset, it->end - it->start);

bool space\_needed = true;

if (it->name.length() > 0) {

space\_needed = false;

line += " " + it->name;

std::string build\_id;

if ((it->flags & PROT\_READ) && elf\_get\_build\_id(backtrace, it->start, &build\_id)) {

line += " (BuildId: " + build\_id + ")";

}

}

if (it->load\_base != 0) {

if (space\_needed) {

line += ' ';

}

line += android::base::StringPrintf(" (load base 0x%" PRIxPTR ")", it->load\_base);

}

\_LOG(log, logtype::MAPS, "%s\n", line.c\_str());

}

if (print\_fault\_address\_marker) {

\_LOG(log, logtype::MAPS, "--->Fault address falls at %s after any mapped regions\n",

get\_addr\_string(addr).c\_str());

}

}

当内存出现故障时，可搜索关键词：

memory map: (fault address prefixed with --->)

### 4.5 dump\_logs

[-> debuggerd/tombstone.cpp]

static void dump\_logs(log\_t\* log, pid\_t pid, unsigned int tail) {

dump\_log\_file(log, pid, "system", tail); //输出system log

dump\_log\_file(log, pid, "main", tail); //输出main log

}

### 4.6 dump\_thread(兄弟)

dump\_thread(log, pid, sibling, map, 0, 0, 0, false);

[-> debuggerd/tombstone.cpp]

static void dump\_thread(log\_t\* log, pid\_t pid, pid\_t tid, BacktraceMap\* map, int signal, int si\_code, uintptr\_t abort\_msg\_address, bool primary\_thread) {

log->current\_tid = tid;

if (!primary\_thread) {

\_LOG(log, logtype::THREAD, "--- --- --- --- --- --- --- --- --- --- --- --- --- --- --- ---\n");

}

//【见小节4.4.1】

dump\_thread\_info(log, pid, tid);

std::unique\_ptr<Backtrace> backtrace(Backtrace::Create(pid, tid, map));

//【见小节4.4.4】

dump\_registers(log, tid);

if (backtrace->Unwind(0)) {

//【见小节4.4.5】

dump\_backtrace\_and\_stack(backtrace.get(), log);

}

log->current\_tid = log->crashed\_tid;

}

兄弟线程dump\_thread的输出内容：

1. 线程相关信息，包含pid/tid以及相应name
2. 寄存器状态
3. backtrace以及stack

### 4.7 小节

**engrave\_tombstone**主要输出信息：

* dump\_header\_info
* 主线程dump\_thread 【见小节4.7.1】
* dump\_logs (ro.debuggable=1 才输出此项)
* 兄弟线程dump\_thread 见小节4.7.2】
* dump\_logs (ro.debuggable=1 才输出此项)

#### 4.7.1 主线程dump\_thread

1. build相关头信息；
2. 线程相关信息，包含pid/tid以及相应name
3. signal相关信息，包含fault address
4. 寄存器状态
5. backtrace
6. stack
7. memory near
8. code around
9. memory map

#### 4.7.2 兄弟线程dump\_thread

1. 线程相关信息，包含pid/tid以及相应name
2. 寄存器状态
3. backtrace
4. stack

兄弟线程调用dump\_thread也会输出的内容其实是主线程dump的第2，4，5，6项目

## 实例

这里是dump\_tombstone文件内容的组成：

#### 5.1 文件头信息

**\*\*\* \*\***\* \***\*\* \*\***\* \***\*\* \*\***\* \***\*\* \*\***\* \***\*\* \*\***\* \***\*\* \*\***\* \***\*\* \*\***\* \***\*\* \*\***\*

//【见小节4.3】dump*\_header\_*info

Build fingerprint: 'xxx/xxx/MMB29M/gityuan06080845:userdebug/test-keys'

Revision: '0'

ABI: 'arm'

#### 5.2 主线程dump\_thread

//【见小节4.4.1】dump\_thread\_info

pid: 1789, tid: 1789, name: system\_server >>> system\_server <<<

//【见小节4.4.2】dump\_signal\_info

signal 19 (SIGSTOP), code 0 (SI\_USER), fault addr --------

//【见小节4.4.4】dump\_registers

r0 fffffffc r1 bed67e68 r2 00000010 r3 0000ea60

r4 00000000 r5 00000008 r6 00000000 r7 0000015a

...

//【见小节4.4.5】dump\_backtrace\_and\_stack

backtrace:

#00 pc 00000000004489bc /data/dalvik-cache/arm64/system@framework@boot.oat (offset 0x3e2e000)

#01 pc 00000000003e8a74 /data/dalvik-cache/arm64/system@framework@boot.oat (offset 0x3e2e000)

stack:

0000007ff47b26b0 0000000012cf05e0 /dev/ashmem/dalvik-main space (deleted)

0000007ff47b26b8 0000000000000000

0000007ff47b26c0 0000000012cf05e0 /dev/ashmem/dalvik-main space (deleted)

//【见小节4.4.6】dump\_memory\_and\_code

memory near r1:

...

code around pc:

code around lr:

//【见小节4.4.7】dump\_all\_maps

memory map:

//【见小节4.5】dump\_logs

#### 5.3 兄弟线程dump\_thread

--- --- --- --- --- --- --- --- --- --- --- --- --- --- --- ---

//【见小节4.4.1】dump\_thread\_info

pid: 1789, tid: 1803, name: Binder\_1 >>> system\_server <<<

//【见小节4.4.4】dump\_registers

r0 0000000b r1 c0186201 r2 b3589868 r3 b3589860

//【见小节4.4.5】dump\_backtrace\_and\_stack

backtrace:

#00 pc 00040aac /system/lib/libc.so (\_\_ioctl+8)

#01 pc 00047529 /system/lib/libc.so (ioctl+14)

#02 pc 0001e909 /system/lib/libbinder.so (\_ZN7android14IPCThreadState14talkWithDriverEb+132)

stack:

b3589810 00000000

b3589814 00000000

b3589818 b6ebf07c /system/lib/libcutils.so

b358981c b6eb4405 /system/lib/libcutils.so

所有兄弟线程是以一系列---作为开头的分割符。

## 六. 总结

这里主要以源码角度来分析debuggerd的原理，整个过程中最重要的产物便是tombstone文件，先留坑，后续再进一步讲述如何分析tombstone文件。

* debuggerd -b <tid>:  
  发送请求的action为DEBUGGER\_ACTION\_DUMP\_BACKTRACE，则调用dump\_backtrace();[Native进程之Trace原理](http://gityuan.com/2016/11/27/native-traces/)
* debuggerd <tid>: 发送请求的action为DEBUGGER\_ACTION\_DUMP\_TOMBSTONE，则调用engrave\_tombstone();
* native crash: 发送请求的action为DEBUGGER\_ACTION\_CRASH，且发送信号为SIGBUS等致命信号，则调用engrave\_tombstone()