作者介绍：王竞原，负责网游刀锋铁骑项目，使用C++10年左右，非常喜欢使用C++，特别是C++11。希望能与广大的C++爱好者多交流。

一、什么是Android的C/C++ NativeCrash

Android上的Crash可以分两种：

1、Java Crash
java代码导致jvm退出，弹出“程序已经崩溃”的对话框，最终用户点击关闭后进程退出。
Logcat 会在“AndroidRuntime”tag下输出Java的调用栈。

2、Native Crash
通过NDK，使用C/C++开发，导致进程收到错误信号，发生Crash，Android 5.0之前进程直接退出（闪退），Android 5.0之后会弹“程序已崩溃”的对话框。

Logcat 会在“debug”tag下输出dump信息：

- 错误信号：11是信号量sigNum，SIGSEGV是信号的名字，SEGV_MAPERR是SIGSEGV下的一种类型。
- 寄存器快照：进程收到错误信号时保存下来的寄存器快照，其中PC寄存器存储的就是下个要运行的指令（出错的位置）。
- 调用栈：#00是栈顶，#02是栈底，#02调用#01调用#00方法，#00的方法时libspirit.so中的Spirit类下的testCrash方法，出错的地方是testCrash方法内汇编偏移17（不是行号哦！）

Tag	Text
test	doCrash
libc	Fatal signal 11 (SIGSEGV), code 1, fault addr 0x0 in tid 1691 (tnativeproject 2)
DEBUG	*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
DEBUG	Build fingerprint: 'google/hammerhead/hammerhead:5.1/LMY47I/1767468:user/release-keys'
DEBUG	Revision: '11'
DEBUG	ABI: 'arm'
DEBUG	pid: 1691, tid: 1691, name: tnativeproject2 >>> com.example.testnativeproject2 <<<
DEBUG	signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 0x0
DEBUG	r0 00000000 r1 ad2208d1 r2 00000001 r3 00000000
DEBUG	r4 b4878ac0 r5 74cc6488 r6 12cdc770 r7 12c29190
DEBUG	r8 12ce6a40 r9 b4827800 sl 00000000 fp 12c29190
DEBUG	ip bee7fba0 sp bee80070 lr b6daedad pc b3509cf6 cpsr 600f0030
DEBUG	backtrace:
DEBUG	#00 pc 00000cf6 /data/app/com.example.testnativeproject2-1/lib/arm/libspirit.so (Spirit::testCrash(_JNIEnv*, _jobject*)+17)
DEBUG	#01 pc 00000d09 /data/app/com.example.testnativeproject2-1/lib/arm/libspirit.so (Java_com_spirit_test_TestNative_testSpirit+4)
DEBUG	#02 pc 00081acb /data/dalvik-cache/arm/data@app@com.example.testnativeproject2-1@base.apk@classes.dex
DEBUG	Tombstone written to: /data/tombstones/tombstone_03

二、什么是错误信号

Android本质就是一个Linux，信号跟Linux信号是同一个东西，信号本身是用于进程间通信的没有正确错误之分，但官方给一些信号赋予了特定的含义及特定处理动作，

通常我们说的错误信号有5个（Bugly全部都能上报），系统默认处理就是dump出堆栈，并退出进程：

信号名	信号量	信号描述
SIGILL	4	机器指令错误，例如给函数指针赋了个非方法来执行时（常见的指针跑飞）
SIGSEGV	11	段错误，非法方式访问了可访问的内存区域（越界访问，写只读内存块等）
SIGBUS	7	内存错误，非法方式访问了不可访问的内存区域（释放一个未被赋值的指针）
SIGFPE	4	算数错误（除0）
SIGABRT	6	主动中止运行

通常的来源有三个：
1、硬件发生异常，即硬件(通常是CPU)检测到一个错误条件并通知Linux内核，内核处理该异常，给相应的进程发送信号。硬件异常的例子包括执行一条异常的机器语言指令，诸如，被0除，或者引用了无法访问的内存区域。大部分信号如果没有被进程处理，默认的操作就是杀死进程。
在本文中，SIGSEGV(段错误)，SIGBUS(内存访问错误)，SIGFPE(算数异常)属于这种信号。

2、进程调用的库发现错误，给自己发送中止信号，默认情况下，该信号会终止进程。在本文中，SIGABRT(中止进程)属于这种信号。

3、用户（手贱）或第三方App（恶意）通过kill-信号 pid的方式给错误进程发送，这时signal中的si_code会小于0。

三、抖几个常见错误

1. 空指针

代码示例

```
int* p = 0; //空指针
*p = 1; //写空指针指向的内存，产生SIGSEGV信号，造成Crash
```

原因分析

在进程的地址空间中，从0开始的第一个页面的权限被设置为不可读也不可写，当进程的指令试图访问该页面中的地址时(如读取空指针指向的内存)，处理器就会产生一个异常，然后Linux内核会给该进程发送一个段错误信号(SIGSEGV)，默认的操作就是杀死进程，并产生core文件。

解决方法

在使用指针前加以判断，如果为空，则是不可访问的。

Bug评述

空指针是很容易出现的一种bug，在代码量大，赶开发进度时很容易出现，但是它也很容易被发现和修复。

2. 野指针

代码示例

```
int* p; //野指针，未初始化，其指向的地址通常是随机的
*p = 1; //写野指针指向的内存，有可能不会马上Crash，而是破坏了别处的内存
```

原因分析

野指针指向的是一个无效的地址，该地址如果是不可读不可写的，那么会马上Crash(内核给进程发送段错误信号SIGSEGV)，这时bug会很快被发现。
如果访问的地址为可写，而且通过野指针修改了该处的内存，那么很有可能会等一段时间(其它的代码使用了该处的内存后)才发生Crash。这时查看Crash时显示的调用栈，和野指针所在的代码部分，有可能基本上没有任何关联。

解决方法

在指针变量定义时，一定要初始化，特别是在结构体或类中的成员指针变量。
在释放了指针指向的内存后，要把该指针置为NULL(但是如果在别的地方也有指针指向该处内存的话，这种方式就不好解决了)。
野指针造成的内存破坏的问题，有时候光看代码很难查找，通过代码分析工具也很难找出，只有通过专业的内存检测工具，才能发现这类bug。

Bug评述

野指针的bug，特别是内存破坏的问题，有时候查起来毫无头绪，没有一点线索，让开发者感觉到很茫然和无助(Bugly上报的堆栈看不出任何问题)。可以说内存破坏bug是服务器稳定性最大的杀手，也是C/C++在开发应用方面相比于其它语言(如Java, C#)的最大劣势之一。

3. 数组越界

代码示例

```
int arr[10];
arr[10] = 1; //数组越界，有可能不会马上Crash，而是破坏了别处的内存
```

原因分析

数组越界和野指针类似，访问了无效的地址，如果该地址不可读写，则会马上Crash(内核给进程发送段错误信号SIGSEGV)，如果修改了该处的内存，造成内存破坏，那么有可能会等一段时间才在别处发生Crash。

解决方法

所有数组遍历的循环，都要加上越界判断。
用下标访问数组时，要判断是否越界。
通过代码分析工具可以发现绝大部分的数组越界问题。

Bug评述

数组越界也是一种内存破坏的bug，有时候与野指针一样也是很难查找的。

4. 整数除以零

代码示例

```
int a = 1;
int b = a / 0; //整数除以0，产生SIGFPE信号，导致Crash
```

原因分析

整数除以零总是产生SIGFPE(浮点异常，产生SIGFPE信号时并非一定要涉及浮点算术，整数运算异常也用浮点异常信号是为了保持向下兼容性)信号，默认的处理方式是终止进程，并生成core文件。

解决方法

在做整数除法时，要判断被除数是否为0的情况。

Bug评述

整数被0除的bug很容易被开发者忽视，因为通常被除数为0的情况在开发环境下很难出现，但是到了生产环境，庞大的用户量和复杂的用户输入，就很容易导致被除数为0的情况出现了。

5. 格式化输出参数错误

代码示例

```
//格式化参数错误，可能会导致非法的内存访问，从而造成宕机
char text[200];
snprintf(text,200,"Valid %u, Invalid %u %s", 1);//format格式不匹配
```

原因分析

格式化参数错误也和野指针类似，但是只会读取无效地址的内存，而不会造成内存破坏，因此其结果是要么打印出错乱的数据，要么访问了无读写权限的内存(收到段错误信号SIGSEGV)而立即宕机。

解决方法

在书写输出格式和参数时，要做到参数个数和类型都要与输出格式一致。
在GCC的编译选项中加入-wformat，让GCC在编译时检测出此类错误。

6、缓冲区溢出

代码示例

```
char szBuffer[10];
//由于函数栈是从高地址往低地址创建，而sprintf是从低地址往高地址打印字符，
//如果超出了缓冲区的大小，函数的栈帧会被破坏，在函数返回时会跳转到未知的地址上，
//基本上都会造成访问异常，从而产生SIGABRT或SIGSEGV，造成Crash

sprintf(szBuffer, "Stack Buffer Overrun!111111111111111"  "111111111111111111111");
```

原因分析

通过往程序的缓冲区写超出其长度的内容，造成缓冲区的溢出，从而破坏函数调用的堆栈，修改函数调用的返回地址。如果不是黑客故意攻击，那么最终函数调用很可能会跳转到无法读写的内存区域，产生段错误信号SIGSEGV或SIGABRT，造成程序崩溃，并生成core文件。

解决方法

检查所有容易产生漏洞的库调用，比如sprintf，strcpy等，它们都没有检查输入参数的长度。
使用带有长度检查的库调用，如用snprintf来代替sprintf，或者自己在sprintf上封装一个带长度检查的函数。
在GCC编译时，在-O1以上的优化行为下，使用-D_FORTIFY_SOURCE=level进行编译(其中level=1或2，level代表的是检测级别的不同，数值越大越严格)。这样GCC会在编译时报告缓冲区溢出的错误。
在GCC编译时加上-fstack-protector或-fstack-protector-all选项，使得堆栈保护(stack-smashingprotector, SSP)功能生效。该功能会在编译后的汇编代码中插入堆栈检测的代码，并在运行时能够检测到栈破坏并输出报告。

Bug评述

缓冲区溢出是一种非常普遍、非常危险的漏洞，在各种操作系统、应用软件中广泛存在。黑客在进行攻击时，输入的字符串一般不会让程序崩溃，而是修改函数的返回地址，使程序跳转到别的地方，转而执行黑客安排好的指令，以达到攻击的目的。
缓冲区溢出后，调试生成的core，可以看见调用栈是混乱的，因为函数的返回地址已经被修改到随机的地址上去了。
服务器宕机后，如果core文件和可执行文件是匹配的，但是调用栈是错乱的，那么很大的可能性是发生了缓冲区溢出。

7、主动抛出异常

代码示例

```
 if ((*env)->ExceptionOccurred(env) != 0) {
        //动态库在内部运行出现错误时，大都会主动abort，终止运行
        abort(); //给当前进程发送信号SIGABRT
 }
```

解决方法

查看堆栈找出abort的原因

Bug评述

如果是程序主动abort的，通过堆栈加源码还是很好定位的，但往往abort的位置是在系统库中，就不好定位了，需要多查看系统API的使用方法，检查是否使用不当。