Diagnosing Native Crashes

If you've never seen a native crash before, start with Debugging Native Android Platform Code (https://source.android.com/devices/tech/debug/index.html).

Types of native crash

The sections below detail the most common kinds of native crash. Each includes an example chunk of `debuggerd` output, with the key evidence that helps you distinguish that specific kind of crash highlighted in orange italic text.

Abort

Aborts are interesting because they're deliberate. There are many different ways to abort (including calling `abort(3)` (http://man7.org/linux/man-pages/man3/abort.3.html), failing an `assert(3)` (http://man7.org/linux/man-pages/man3/assert.3.html), using one of the Android-specific fatal logging types), but they all involve calling `abort`. A call to `abort` basically signals the calling thread with SIGABRT, so a frame showing "abort" in `libc.so` plus SIGABRT are the things to look for in the `debuggerd` output to recognize this case.

As mentioned above, there may be an explicit "abort message" line. But you should also look in the `logcat` output to see what this thread logged before deliberately killing itself, because the basic abort primitive doesn't accept a message.

Older versions of Android (especially on 32-bit ARM) followed a convoluted path between the original abort call (frame 4 here) and the actual sending of the signal (frame 0 here):

```
pid: 1656, tid: 1656, name: crasher  >>> crasher <<<
signal 6 (SIGABRT), code -6 (SI_TKILL), fault addr --------
Abort message: 'some_file.c:123: some_function: assertion "false" failed'
    r0 00000000  r1 00000678  r2 00000006  r3 f70b6dc8
    r4 f70b6dd0  r5 f70b6d80  r6 00000002  r7 0000010c
    r8 ffffffed  r9 00000000  sl 00000000  fp ff96ae1c
    ip 00000006  sp ff96ad18  lr f700ced5  pc f700dc98  cpsr 400b0010
backtrace:
    #00 pc 00042c98  /system/lib/libc.so (tgkill+12)
    #01 pc 00041ed1  /system/lib/libc.so (pthread_kill+32)
    #02 pc 0001bb87  /system/lib/libc.so (raise+10)
    #03 pc 00018cad  /system/lib/libc.so (__libc_android_abort+34)
    #04 pc 000168e8  /system/lib/libc.so (abort+4)
    #05 pc 0001a78f  /system/lib/libc.so (__libc_fatal+16)
    #06 pc 00018d35  /system/lib/libc.so (__assert2+20)
    #07 pc 00000f21  /system/xbin/crasher
    #08 pc 00016795  /system/lib/libc.so (__libc_init+44)
    #09 pc 00000abc  /system/xbin/crasher
```

More recent versions call `tgkill(2)` (http://man7.org/linux/man-pages/man2/tgkill.2.html) directly from `abort`, so there are fewer stack frames for you to skip over before you get to the interesting frames:

```
pid: 25301, tid: 25301, name: crasher  >>> crasher <<<
signal 6 (SIGABRT), code -6 (SI_TKILL), fault addr --------
    r0 00000000  r1 000062d5  r2 00000006  r3 00000008
    r4 ffa09dd8  r5 000062d5  r6 000062d5  r7 0000010c
    r8 00000000  r9 00000000  sl 00000000  fp ffa09f0c
    ip 00000000  sp ffa09dc8  lr eac63ce3  pc eac93f0c  cpsr 000d0010
backtrace:
    #00 pc 00049f0c  /system/lib/libc.so (tgkill+12)
    #01 pc 00019cdf  /system/lib/libc.so (abort+50)
    #02 pc 000012db  /system/xbin/crasher (maybe_abort+26)
    #03 pc 000015b7  /system/xbin/crasher (do_action+414)
    #04 pc 000020d5  /system/xbin/crasher (main+100)
    #05 pc 000177a1  /system/lib/libc.so (__libc_init+48)
    #06 pc 000010e4  /system/xbin/crasher (_start+96)
```

You can reproduce an instance of this type of crash using: `crasher abort`

Pure null pointer dereference

This is the classic native crash, and although it's just a special case of the next crash type, it's worth mentioning separately because it usually requires the least thought.

In the example below, even though the crashing function is in `libc.so`, because the string functions just operate on the pointers they're given, you can infer that **`strlen(3)`** (http://man7.org/linux/man-pages/man3/strlen.3.html) was called with a null pointer; and this crash should go straight to the author of the calling code. In this case, frame #01 is the bad caller.

```
pid: 25326, tid: 25326, name: crasher  >>> crasher <<<
signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 0x0
    r0 00000000  r1 00000000  r2 00004c00  r3 00000000
    r4 ab088071  r5 fff92b34  r6 00000002  r7 fff92b40
    r8 00000000  r9 00000000  sl 00000000  fp fff92b2c
    ip ab08cfc4  sp fff92a08  lr ab087a93  pc efb78988  cpsr 600d0030

backtrace:
    #00 pc 00019988  /system/lib/libc.so (strlen+71)
    #01 pc 00001a8f  /system/xbin/crasher (strlen_null+22)
    #02 pc 000017cd  /system/xbin/crasher (do_action+948)
    #03 pc 000020d5  /system/xbin/crasher (main+100)
    #04 pc 000177a1  /system/lib/libc.so (__libc_init+48)
    #05 pc 000010e4  /system/xbin/crasher (_start+96)
```

You can reproduce an instance of this type of crash using: `crasher strlen-NULL`

Low-address null pointer dereference

In many cases the fault address won't be 0, but some other low number. Two- or three-digit addresses in particular are very common, whereas a six-digit address is almost certainly not a null pointer dereference—that would require a 1MiB offset. This usually occurs when you have code that dereferences a null pointer as if it was a valid struct. Common functions are **`fprintf(3)`** (http://man7.org/linux/man-pages/man3/fprintf.3.html) (or any other function taking a FILE*) and **`readdir(3)`** (http://man7.org/linux/man-pages/man3/readdir.3.html), because code often fails to check that the **`fopen(3)`** (http://man7.org/linux/man-pages/man3/fopen.3.html) or **`opendir(3)`** (http://man7.org/linux/man-pages/man3/opendir.3.html) call actually succeeded first.

Here's an example of `readdir`:

```
pid: 25405, tid: 25405, name: crasher  >>> crasher <<<
signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 0xc
    r0 0000000c  r1 00000000  r2 00000000  r3 3d5f0000
    r4 00000000  r5 0000000c  r6 00000002  r7 ff8618f0
    r8 00000000  r9 00000000  sl 00000000  fp ff8618dc
    ip edaa6834  sp ff8617a8  lr eda34a1f  pc eda618f6  cpsr 600d0030

backtrace:
    #00 pc 000478f6  /system/lib/libc.so (pthread_mutex_lock+1)
    #01 pc 0001aa1b  /system/lib/libc.so (readdir+10)
    #02 pc 00001b35  /system/xbin/crasher (readdir_null+20)
    #03 pc 00001815  /system/xbin/crasher (do_action+976)
    #04 pc 000021e5  /system/xbin/crasher (main+100)
    #05 pc 000177a1  /system/lib/libc.so (__libc_init+48)
    #06 pc 00001110  /system/xbin/crasher (_start+96)
```

Here the direct cause of the crash is that **`pthread_mutex_lock(3)`** (http://man7.org/linux/man-pages/man3/pthread_mutex_lock.3p.html) has tried to access address 0xc (frame 0). But the first thing `pthread_mutex_lock` does is dereference the `state` element of the `pthread_mutex_t*` it was given. If you look at the source, you can see that element is at offset 0 in the struct, which tells you that `pthread_mutex_lock` was given the invalid pointer 0xc. From the frame 1 you can see that it was given that pointer by `readdir`, which extracts the `mutex_` field from the `DIR*` it's given. Looking at that structure, you can see that `mutex_` is at offset `sizeof(int) + sizeof(size_t) + sizeof(dirent*)` into `struct DIR`, which on a 32-bit device is 4 + 4 + 4 = 12 = 0xc, so you found the bug: `readdir` was passed a null pointer by the caller. At this point you can paste the stack into the stack tool to find out *where* in logcat this happened.

```
  struct DIR {
    int fd_;
    size_t available_bytes_;
```

```
    dirent* next_;
    pthread_mutex_t mutex_;
    dirent buff_[15];
    long current_pos_;
  };
```

In most cases you can actually skip this analysis. A sufficiently low fault address usually means you can just skip any `libc.so` frames in the stack and directly accuse the calling code. But not always, and this is how you would present a compelling case.

You can reproduce instances of this kind of crash using: `crasher fprintf-NULL` or `crasher readdir-NULL`

FORTIFY failure

A FORTIFY failure is a special case of an abort that occurs when the C library detects a problem that might lead to a security vulnerability. Many C library functions are *fortified*; they take an extra argument that tells them how large a buffer actually is and check at run time whether the operation you're trying to perform actually fits. Here's an example where the code tries to `read(fd, buf, 32)` into a buffer that's actually only 10 bytes long...

```
pid: 25579, tid: 25579, name: crasher  >>> crasher <<<
signal 6 (SIGABRT), code -6 (SI_TKILL), fault addr --------
Abort message: 'FORTIFY: read: prevented 32-byte write into 10-byte buffer'
    r0 00000000  r1 000063eb  r2 00000006  r3 00000008
    r4 ff96f350  r5 000063eb  r6 000063eb  r7 0000010c
    r8 00000000  r9 00000000  sl 00000000  fp ff96f49c
    ip 00000000  sp ff96f340  lr ee83ece3  pc ee86ef0c  cpsr 000d0010

backtrace:
    #00 pc 00049f0c  /system/lib/libc.so (tgkill+12)
    #01 pc 00019cdf  /system/lib/libc.so (abort+50)
    #02 pc 0001e197  /system/lib/libc.so (__fortify_fatal+30)
    #03 pc 0001baf9  /system/lib/libc.so (__read_chk+48)
    #04 pc 0000165b  /system/xbin/crasher (do_action+534)
    #05 pc 000021e5  /system/xbin/crasher (main+100)
    #06 pc 000177a1  /system/lib/libc.so (__libc_init+48)
    #07 pc 00001110  /system/xbin/crasher (_start+96)
```

You can reproduce an instance of this type of crash using: `crasher fortify`

Stack corruption detected by -fstack-protector

The compiler's `-fstack-protector` option inserts checks into functions with on-stack buffers to guard against buffer overruns. This option is on by default for platform code but not for apps. When this option is enabled, the compiler adds instructions to the function prologue (https://en.wikipedia.org/wiki/Function_prologue) to write a random value just past the last local on the stack and to the function epilogue to read it back and check that it's not changed. If that value changed, it was overwritten by a buffer overrun, so the epilogue calls `__stack_chk_fail` to log a message and abort.

```
pid: 26717, tid: 26717, name: crasher  >>> crasher <<<
signal 6 (SIGABRT), code -6 (SI_TKILL), fault addr --------
Abort message: 'stack corruption detected'
    r0 00000000  r1 0000685d  r2 00000006  r3 00000008
    r4 ffd516d8  r5 0000685d  r6 0000685d  r7 0000010c
    r8 00000000  r9 00000000  sl 00000000  fp ffd518bc
    ip 00000000  sp ffd516c8  lr ee63ece3  pc ee66ef0c  cpsr 000e0010

backtrace:
    #00 pc 00049f0c  /system/lib/libc.so (tgkill+12)
    #01 pc 00019cdf  /system/lib/libc.so (abort+50)
    #02 pc 0001e07d  /system/lib/libc.so (__libc_fatal+24)
    #03 pc 0004863f  /system/lib/libc.so (__stack_chk_fail+6)
    #04 pc 000013ed  /system/xbin/crasher (smash_stack+76)
    #05 pc 00001591  /system/xbin/crasher (do_action+280)
    #06 pc 00002219  /system/xbin/crasher (main+100)
    #07 pc 000177a1  /system/lib/libc.so (__libc_init+48)
    #08 pc 00001144  /system/xbin/crasher (_start+96)
```

You can distinguish this from other kinds of abort by the presence of `__stack_chk_fail` in the backtrace and the specific abort message.

You can reproduce an instance of this type of crash using: `crasher smash-stack`

Crash dumps

If you don't have a specific crash that you're investigating right now, the platform source includes a tool for testing **`debuggerd`** called crasher. If you **`mm`** in **`system/core/debuggerd/`** you'll get both a **`crasher`** and a **`crasher64`** on your path (the latter allowing you to test 64-bit crashes). Crasher can crash in a large number of interesting ways based on the command line arguments you provide. Use **`crasher --help`** to see the currently supported selection.

To introduce the different pieces in a crash dump, let's work through this example crash dump:

```
*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
Build fingerprint: 'Android/aosp_flounder/flounder:5.1.51/AOSP/enh08201009:eng/test-keys'
Revision: '0'
ABI: 'arm'
pid: 1656, tid: 1656, name: crasher  >>> crasher <<<
signal 6 (SIGABRT), code -6 (SI_TKILL), fault addr --------
Abort message: 'some_file.c:123: some_function: assertion "false" failed'
    r0 00000000  r1 00000678  r2 00000006  r3 f70b6dc8
    r4 f70b6dd0  r5 f70b6d80  r6 00000002  r7 0000010c
    r8 ffffffed  r9 00000000  sl 00000000  fp ff96ae1c
    ip 00000006  sp ff96ad18  lr f700ced5  pc f700dc98  cpsr 400b0010
backtrace:
    #00 pc 00042c98  /system/lib/libc.so (tgkill+12)
    #01 pc 00041ed1  /system/lib/libc.so (pthread_kill+32)
    #02 pc 0001bb87  /system/lib/libc.so (raise+10)
    #03 pc 00018cad  /system/lib/libc.so (__libc_android_abort+34)
    #04 pc 000168e8  /system/lib/libc.so (abort+4)
    #05 pc 0001a78f  /system/lib/libc.so (__libc_fatal+16)
    #06 pc 00018d35  /system/lib/libc.so (__assert2+20)
    #07 pc 00000f21  /system/xbin/crasher
    #08 pc 00016795  /system/lib/libc.so (__libc_init+44)
    #09 pc 00000abc  /system/xbin/crasher
Tombstone written to: /data/tombstones/tombstone_06
```

```
*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
```

The line of asterisks with spaces is helpful if you're searching a log for native crashes. The string "*** ***" rarely shows up in logs other than at the beginning of a native crash.

```
Build fingerprint:
'Android/aosp_flounder/flounder:5.1.51/AOSP/enh08201009:eng/test-keys'
```

The fingerprint lets you identify exactly which build the crash occurred on. This is exactly the same as the **`ro.build.fingerprint`** system property.

```
Revision: '0'
```

The revision refers to the hardware rather than the software. This is usually unused but can be useful to help you automatically ignore bugs known to be caused by bad hardware. This is exactly the same as the **`ro.revision`** system property.

```
ABI: 'arm'
```

The ABI is one of arm, arm64, mips, mips64, x86, or x86-64. This is mostly useful for the **`stack`** script mentioned above, so that it knows what toolchain to use.

```
pid: 1656, tid: 1656, name: crasher >>> crasher <<<
```

This line identifies the specific thread in the process that crashed. In this case, it was the process' main thread, so the process ID and thread ID match. The first name is the thread name, and the name surrounded by >>> and <<< is the process name. For an app, the process name is typically the fully-qualified package name (such as com.facebook.katana), which is useful when filing bugs or trying to find the app in Google Play. The pid and tid can also be useful in finding the relevant log lines preceding the crash.

```
signal 6 (SIGABRT), code -6 (SI_TKILL), fault addr --------
```

This line tells you which signal (SIGABRT) was received, and more about how it was received (SI_TKILL). The signals reported by **debuggerd** are SIGABRT, SIGBUS, SIGFPE, SIGILL, SIGSEGV, and SIGTRAP. The signal-specific codes vary based on the specific signal.

```
Abort message: 'some_file.c:123: some_function: assertion "false" failed'
```

Not all crashes will have an abort message line, but aborts will. This is automatically gathered from the last line of fatal logcat output for this pid/tid, and in the case of a deliberate abort is likely to give an explanation of why the program killed itself.

```
r0 00000000 r1 00000678 r2 00000006 r3 f70b6dc8
r4 f70b6dd0 r5 f70b6d80 r6 00000002 r7 0000010c
r8 ffffffed r9 00000000 sl 00000000 fp ff96ae1c
ip 00000006 sp ff96ad18 lr f700ced5 pc f700dc98 cpsr 400b0010
```

The register dump shows the content of the CPU registers at the time the signal was received. (This section varies wildly between ABIs.) How useful these are will depend on the exact crash.

```
backtrace:
    #00 pc 00042c98 /system/lib/libc.so (tgkill+12)
    #01 pc 00041ed1 /system/lib/libc.so (pthread_kill+32)
    #02 pc 0001bb87 /system/lib/libc.so (raise+10)
    #03 pc 00018cad /system/lib/libc.so (__libc_android_abort+34)
    #04 pc 000168e8 /system/lib/libc.so (abort+4)
    #05 pc 0001a78f /system/lib/libc.so (__libc_fatal+16)
    #06 pc 00018d35 /system/lib/libc.so (__assert2+20)
    #07 pc 00000f21 /system/xbin/crasher
    #08 pc 00016795 /system/lib/libc.so (__libc_init+44)
    #09 pc 00000abc /system/xbin/crasher
```

The backtrace shows you where in the code we were at the time of crash. The first column is the frame number (matching gdb's style where the deepest frame is 0). The PC values are relative to the location of the shared library rather than absolute addresses. The next column is the name of the mapped region (which is usually a shared library or executable, but might not be for, say, JIT-compiled code). Finally, if symbols are available, the symbol that the PC value corresponds to is shown, along with the offset into that symbol in bytes. You can use this in conjunction with `objdump(1)` to find the corresponding assembler instruction.

Tombstones

```
Tombstone written to: /data/tombstones/tombstone_06
```

This tells you where **debuggerd** wrote extra information. **debuggerd** will keep up to 10 tombstones, cycling through the numbers 00 to 09 and overwriting existing tombstones as necessary.

The tombstone contains the same information as the crash dump, plus a few extras. For example, it includes backtraces for *all* threads (not just the crashing thread), the floating point registers, raw stack dumps, and memory dumps around the addresses in registers. Most usefully it also includes a full memory map (similar to **/proc/*pid*/maps**). Here's an annotated example from a 32-bit ARM process crash:

```
memory map: (fault address prefixed with --->)
--->ab15f000-ab162fff r-x 0 4000 /system/xbin/crasher (BuildId:
b9527db01b5cf8f5402f899f64b9b121)
```

There are two things to note here. The first is that this line is prefixed with "--->". The maps are most useful when your crash isn't just a null pointer dereference. If the fault address is small, it's probably some variant of a null pointer dereference. Otherwise looking at the maps around the fault address can often give you a clue as to what happened. Some possible issues that can be recognized by looking at the maps include:

- Reads/writes past the end of a block of memory.
- Reads/writes before the beginning of a block of memory.
- Attempts to execute non-code.
- Running off the end of a stack.
- Attempts to write to code (as in the example above).

The second thing to note is that executables and shared libraries files will show the BuildId (if present) in Android M and later, so you can see exactly which version of your code crashed. (Platform binaries include a BuildId by default since Android M. NDK r12 and later automatically pass `-Wl,--build-id` to the linker too.)

```
ab163000-ab163fff r--      3000      1000  /system/xbin/crasher
ab164000-ab164fff rw-         0      1000
f6c80000-f6d7ffff rw-         0    100000  [anon:libc_malloc]
```

On Android the heap isn't necessarily a single region. Heap regions will be labeled `[anon:libc_malloc]`.

```
f6d82000-f6da1fff r--         0     20000  /dev/__properties__/u:object_r:logd_prop:s0
f6da2000-f6dc1fff r--         0     20000  /dev/__properties__/u:object_r:default_prop:s0
f6dc2000-f6de1fff r--         0     20000  /dev/__properties__/u:object_r:logd_prop:s0
f6de2000-f6de5fff r-x         0      4000  /system/lib/libnetd_client.so (BuildId: 08020aa06ed48cf9f6971861abf06c9d)
f6de6000-f6de6fff r--      3000      1000  /system/lib/libnetd_client.so
f6de7000-f6de7fff rw-      4000      1000  /system/lib/libnetd_client.so
f6dec000-f6e74fff r-x         0     89000  /system/lib/libc++.so (BuildId: 8f1f2be4b37d7067d366543fafececa2) (load ba
f6e75000-f6e75fff ---         0      1000
f6e76000-f6e79fff r--     89000      4000  /system/lib/libc++.so
f6e7a000-f6e7afff rw-     8d000      1000  /system/lib/libc++.so
f6e7b000-f6e7bfff rw-         0      1000  [anon:.bss]
f6e7c000-f6efdfff r-x         0     82000  /system/lib/libc.so (BuildId: d189b369d1aafe11feb7014d411bb9c3)
f6efe000-f6f01fff r--     81000      4000  /system/lib/libc.so
f6f02000-f6f03fff rw-     85000      2000  /system/lib/libc.so
f6f04000-f6f04fff rw-         0      1000  [anon:.bss]
f6f05000-f6f05fff r--         0      1000  [anon:.bss]
f6f06000-f6f0bfff rw-         0      6000  [anon:.bss]
f6f0c000-f6f21fff r-x         0     16000  /system/lib/libcutils.so (BuildId: d6d68a419dadd645ca852cd339f89741)
f6f22000-f6f22fff r--     15000      1000  /system/lib/libcutils.so
f6f23000-f6f23fff rw-     16000      1000  /system/lib/libcutils.so
f6f24000-f6f31fff r-x         0      e000  /system/lib/liblog.so (BuildId: e4d30918d1b1028a1ba23d2ab72536fc)
f6f32000-f6f32fff r--      d000      1000  /system/lib/liblog.so
f6f33000-f6f33fff rw-      e000      1000  /system/lib/liblog.so
```

Typically a shared library will have three adjacent entries. One will be readable and executable (code), one will be read-only (read-only data), and one will be read-write (mutable data). The first column shows the address ranges for the mapping, the second column the permissions (in the usual Unix `ls(1)` style), the third column the offset into the file (in hex), the fourth column the size of the region (in hex), and the fifth column the file (or other region name).

```
f6f34000-f6f53fff r-x         0     20000  /system/lib/libm.so (BuildId: 76ba45dcd9247e60227200976a02c69b)
f6f54000-f6f54fff ---         0      1000
f6f55000-f6f55fff r--     20000      1000  /system/lib/libm.so
f6f56000-f6f56fff rw-     21000      1000  /system/lib/libm.so
f6f58000-f6f58fff rw-         0      1000
f6f59000-f6f78fff r--         0     20000  /dev/__properties__/u:object_r:default_prop:s0
f6f79000-f6f98fff r--         0     20000  /dev/__properties__/properties_serial
f6f99000-f6f99fff rw-         0      1000  [anon:linker_alloc_vector]
f6f9a000-f6f9afff r--         0      1000  [anon:atexit handlers]
f6f9b000-f6fbafff r--         0     20000  /dev/__properties__/properties_serial
f6fbb000-f6fbbfff rw-         0      1000  [anon:linker_alloc_vector]
f6fbc000-f6fbcfff rw-         0      1000  [anon:linker_alloc_small_objects]
f6fbd000-f6fbdfff rw-         0      1000  [anon:linker_alloc_vector]
f6fbe000-f6fbffff rw-         0      2000  [anon:linker_alloc]
f6fc0000-f6fc0fff r--         0      1000  [anon:linker_alloc]
f6fc1000-f6fc1fff rw-         0      1000  [anon:linker_alloc_lob]
f6fc2000-f6fc2fff r--         0      1000  [anon:linker_alloc]
f6fc3000-f6fc3fff rw-         0      1000  [anon:linker_alloc_vector]
f6fc4000-f6fc4fff rw-         0      1000  [anon:linker_alloc_small_objects]
f6fc5000-f6fc5fff rw-         0      1000  [anon:linker_alloc_vector]
f6fc6000-f6fc6fff rw-         0      1000  [anon:linker_alloc_small_objects]
f6fc7000-f6fc7fff rw-         0      1000  [anon:arc4random _rsx structure]
f6fc8000-f6fc8fff rw-         0      1000  [anon:arc4random _rs structure]
f6fc9000-f6fc9fff r--         0      1000  [anon:atexit handlers]
f6fca000-f6fcafff ---         0      1000  [anon:thread signal stack guard page]
```

Note that since Android 5.0 (Lollipop), the C library names most of its anonymous mapped regions so there are fewer mystery regions.

```
f6fcb000-f6fccfff rw- 0 2000 [stack:5081]
```

Regions named `[stack:`*`tid`*`]` are the stacks for the given threads.

```
f6fcd000-f702afff r-x         0     5e000  /system/bin/linker (BuildId: 84f1316198deee0591c8ac7f158f28b7)
f702b000-f702cfff r--     5d000      2000  /system/bin/linker
f702d000-f702dfff rw-     5f000      1000  /system/bin/linker
f702e000-f702ffff rw-         0      2000
f7030000-f7030fff r--         0      1000
f7031000-f7032fff rw-         0      2000
ffcd7000-ffcf7fff rw-         0     21000
ffff0000-ffff0fff r-x         0      1000  [vectors]
```

Whether you see `[vector]` or `[vdso]` depends on the architecture. ARM uses [vector], while all other architectures use [vdso]. (http://man7.org/linux/man-pages/man7/vdso.7.html)

Last updated March 27, 2017.