|  |  |  |
| --- | --- | --- |
|  | **2, LEBUH ACHEH, GEORGE TOWN**  **10300 GEORGE TOWN**  **PULAU PINANG**  **INFORMATION SHEET** | |
| **PROGRAM’S CODE & NAME** | J620-002-4:2020 FRONT-END SOFTWARE DEVELOPMENT | |
| **LEVEL** | FOUR (4) | |
| **COMPETENCY UNIT NO. AND TITLE** | J620-002-4:2020-C04: MOBILE APPLICATION WITH THIRD PARTY API DEVELOPMENT | |
| **WORK ACTIVITIES NO. AND STATEMENT** | 1. CREATE MOBILE APP DESIGN MOCK-UP ELEMENTS. 2. PLAN MOBILE APP DESIGN STRUCTURE. 3. TRANSFORM MOCK-UP TO MOBILE APP. 4. INTEGRATE MOBILE APP WITH DATA SOURCE. 5. VERIFY SUCCESSFUL API INTEGRATION 6. **VERIFY DEVELOPED MOBILE APP.** 7. VERIFY MOBILE APP ACCESSIBLE GLOBALLY. | |
| **CODE NO.** | J620-002-4:2020-C04/IS(12/15) | Page: 1 of |

**TITLE**:

**DEBUGGING AND LOGGING**

**PURPOSE**:

This information sheet is intended to provide insight and knowledge to trainees with regards to the fundamentals of debugging and logging.

**INFORMATION:**

This information sheet provides useful notes and explanations on fundamental concepts of debugging and logging.

# **DEBUGGING NATIVE ANDROID PLATFORM CODE**

## Crash dumps and tombstones

When a dynamically linked executable starts, several signal handlers are registered that, in the event of a crash, cause a basic crash dump to be written to logcat and a more detailed tombstone file to be written to /data/tombstones/. The tombstone is a file with extra data about the crashed process. In particular, it contains stack traces for all the threads in the crashing process (not just the thread that caught the signal), a full memory map, and a list of all open file descriptors.

Before Android 8.0, crashes were handled by the debuggerd and debuggerd64 daemons. In Android 8.0 and higher, crash\_dump32 and crash\_dump64 are spawned as needed.

It's possible for the crash dumper to attach only if nothing else is already attached, which means that using tools such as strace or lldb prevent crash dumps from occurring.

Example output (with timestamps and extraneous information removed):

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

Build fingerprint: 'Android/aosp\_angler/angler:7.1.1/NYC/enh12211018:eng/test-keys'

Revision: '0'

ABI: 'arm'

pid: 17946, tid: 17949, name: crasher >>> crasher <<<

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

r0 0000000c r1 00000000 r2 00000000 r3 00000000

r4 00000000 r5 0000000c r6 eccdd920 r7 00000078

r8 0000461a r9 ffc78c19 sl ab209441 fp fffff924

ip ed01b834 sp eccdd800 lr ecfa9a1f pc ecfd693e cpsr 600e0030

backtrace:

#00 pc 0004793e /system/lib/libc.so (pthread\_mutex\_lock+1)

#01 pc 0001aa1b /system/lib/libc.so (readdir+10)

#02 pc 00001b91 /system/xbin/crasher (readdir\_null+20)

#03 pc 0000184b /system/xbin/crasher (do\_action+978)

#04 pc 00001459 /system/xbin/crasher (thread\_callback+24)

#05 pc 00047317 /system/lib/libc.so (\_ZL15\_\_pthread\_startPv+22)

#06 pc 0001a7e5 /system/lib/libc.so (\_\_start\_thread+34)

Tombstone written to: /data/tombstones/tombstone\_06

The last line of output gives the location of the full tombstone on disk.

If you have the unstripped binaries available, you can get a more detailed unwind with line number information by pasting the stack into development/scripts/stack:

development/scripts/stack

Example output (based on the logcat output above):

Reading native crash info from stdin

03-02 23:53:49.477 17951 17951 F DEBUG : \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\*

03-02 23:53:49.477 17951 17951 F DEBUG : Build fingerprint: 'Android/aosp\_angler/angler:7.1.1/NYC/enh12211018:eng/test-keys'

03-02 23:53:49.477 17951 17951 F DEBUG : Revision: '0'

03-02 23:53:49.477 17951 17951 F DEBUG : ABI: 'arm'

03-02 23:53:49.478 17951 17951 F DEBUG : pid: 17946, tid: 17949, name: crasher >>> crasher <<<

03-02 23:53:49.478 17951 17951 F DEBUG : signal 11 (SIGSEGV), code 1 (SEGV\_MAPERR), fault addr 0xc

03-02 23:53:49.478 17951 17951 F DEBUG : r0 0000000c r1 00000000 r2 00000000 r3 00000000

03-02 23:53:49.478 17951 17951 F DEBUG : r4 00000000 r5 0000000c r6 eccdd920 r7 00000078

03-02 23:53:49.478 17951 17951 F DEBUG : r8 0000461a r9 ffc78c19 sl ab209441 fp fffff924

03-02 23:53:49.478 17951 17951 F DEBUG : ip ed01b834 sp eccdd800 lr ecfa9a1f pc ecfd693e cpsr 600e0030

03-02 23:53:49.491 17951 17951 F DEBUG :

03-02 23:53:49.491 17951 17951 F DEBUG : backtrace:

03-02 23:53:49.492 17951 17951 F DEBUG : #00 pc 0004793e /system/lib/libc.so (pthread\_mutex\_lock+1)

03-02 23:53:49.492 17951 17951 F DEBUG : #01 pc 0001aa1b /system/lib/libc.so (readdir+10)

03-02 23:53:49.492 17951 17951 F DEBUG : #02 pc 00001b91 /system/xbin/crasher (readdir\_null+20)

03-02 23:53:49.492 17951 17951 F DEBUG : #03 pc 0000184b /system/xbin/crasher (do\_action+978)

03-02 23:53:49.492 17951 17951 F DEBUG : #04 pc 00001459 /system/xbin/crasher (thread\_callback+24)

03-02 23:53:49.492 17951 17951 F DEBUG : #05 pc 00047317 /system/lib/libc.so (\_ZL15\_\_pthread\_startPv+22)

03-02 23:53:49.492 17951 17951 F DEBUG : #06 pc 0001a7e5 /system/lib/libc.so (\_\_start\_thread+34)

03-02 23:53:49.492 17951 17951 F DEBUG : Tombstone written to: /data/tombstones/tombstone\_06

Reading symbols from /huge-ssd/aosp-arm64/out/target/product/angler/symbols

Revision: '0'

pid: 17946, tid: 17949, name: crasher >>> crasher <<<

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

r0 0000000c r1 00000000 r2 00000000 r3 00000000

r4 00000000 r5 0000000c r6 eccdd920 r7 00000078

r8 0000461a r9 ffc78c19 sl ab209441 fp fffff924

ip ed01b834 sp eccdd800 lr ecfa9a1f pc ecfd693e cpsr 600e0030

Using arm toolchain from: /huge-ssd/aosp-arm64/prebuilts/gcc/linux-x86/arm/arm-linux-androideabi-4.9/bin/

Stack Trace:

RELADDR FUNCTION FILE:LINE

0004793e pthread\_mutex\_lock+2 bionic/libc/bionic/pthread\_mutex.cpp:515

v------> ScopedPthreadMutexLocker bionic/libc/private/ScopedPthreadMutexLocker.h:27

0001aa1b readdir+10 bionic/libc/bionic/dirent.cpp:120

00001b91 readdir\_null+20 system/core/debuggerd/crasher.cpp:131

0000184b do\_action+978 system/core/debuggerd/crasher.cpp:228

00001459 thread\_callback+24 system/core/debuggerd/crasher.cpp:90

00047317 \_\_pthread\_start(void\*)+22 bionic/libc/bionic/pthread\_create.cpp:202 (discriminator 1)

0001a7e5 \_\_start\_thread+34 bionic/libc/bionic/clone.cp

You can use stack on an entire tombstone. Example:

stack < FS/data/tombstones/tombstone\_05

This is useful if you've just unzipped a bug report in the current directory. For more information about diagnosing native crashes and tombstones, see Diagnosing Native Crashes.

## Getting a stack trace/tombstone from a running process

You can use the debuggerd tool to get a stack dump from a running process. From the command line, invoke debuggerd using a process ID (PID) to dump a full tombstone to stdout. To get just the stack for every thread in the process, include the -b or --backtrace flag.

## Understanding a complex unwind

When an app crashes, the stack tends to be pretty complex. The following detailed example highlights many of the complexities:

#00 pc 00000000007e6918 /system/priv-app/Velvet/Velvet.apk (offset

0x346b000)

#01 pc 00000000001845cc /system/priv-app/Velvet/Velvet.apk (offset 0x346b000)

#02 pc 00000000001847e4 /system/priv-app/Velvet/Velvet.apk (offset 0x346b000)

#03 pc 00000000001805c0 /system/priv-app/Velvet/Velvet.apk (offset 0x346b000) (Java\_com\_google\_speech\_recognizer\_AbstractRecognizer\_nativeRun+176)

Frames #00–#03 are from native JNI code that was stored uncompressed in the APK to save disk space rather than being extracted into a separate .so file. The stack unwinder in Android 9 and higher doesn’t need the extracted .so file to cope with this common Android-specific case.

Frames #00–#02 don’t have symbol names because they were stripped by the developer.

Frame #03 shows that where symbols are available, the unwinder uses them.

#04 pc 0000000000117550 /data/dalvik-cache/arm64/system@priv-app@Velvet@Velvet.apk@classes.dex (offset 0x108000) (com.google.speech.recognizer.AbstractRecognizer.nativeRun+160)

Frame #04 is ahead-of-time compiled Java code. The old unwinder would have stopped here, unable to unwind through Java.

#05 pc 0000000000559f88 /system/lib64/libart.so (art\_quick\_invoke\_stub+584)

#06 pc 00000000000ced40 /system/lib64/libart.so (art::ArtMethod::Invoke(art::Thread\*, unsigned int\*, unsigned int, art::JValue\*, char const\*)+200)

#07 pc 0000000000280cf0 /system/lib64/libart.so (art::interpreter::ArtInterpreterToCompiledCodeBridge(art::Thread\*, art::ArtMethod\*, art::ShadowFrame\*, unsigned short, art::JValue\*)+344)

#08 pc 000000000027acac /system/lib64/libart.so (bool art::interpreter::DoCall<false, false>(art::ArtMethod\*, art::Thread\*, art::ShadowFrame&, art::Instruction const\*, unsigned short, art::JValue\*)+948)

#09 pc 000000000052abc0 /system/lib64/libart.so (MterpInvokeDirect+296)

#10 pc 000000000054c614 /system/lib64/libart.so (ExecuteMterpImpl+14484)

Frames #05–#10 are from the ART interpreter implementation. The stack unwinder in releases lower than Android 9 would have shown these frames without the context of frame #11 explaining what code the interpreter was interpreting. These frames are useful if you're debugging ART itself. If you're debugging an app, you can ignore them. Some tools, such as simpleperf, automatically omit these frames.

#11 pc 00000000001992d6 /system/priv-app/Velvet/Velvet.apk (offset 0x26cf000) (com.google.speech.recognizer.AbstractRecognizer.run+18)

Frame #11 is the Java code being interpreted.

#12 pc 00000000002547a8 /system/lib64/libart.so (\_ZN3art11interpreterL7ExecuteEPNS\_6ThreadERKNS\_20CodeItemDataAccessorERNS\_11ShadowFrameENS\_6JValueEb.llvm.780698333+496)

#13 pc 000000000025a328 /system/lib64/libart.so (art::interpreter::ArtInterpreterToInterpreterBridge(art::Thread\*, art::CodeItemDataAccessor const&, art::ShadowFrame\*, art::JValue\*)+216)

#14 pc 000000000027ac90 /system/lib64/libart.so (bool art::interpreter::DoCall<false, false>(art::ArtMethod\*, art::Thread\*, art::ShadowFrame&, art::Instruction const\*, unsigned short, art::JValue\*)+920)

#15 pc 0000000000529880 /system/lib64/libart.so (MterpInvokeVirtual+584)

#16 pc 000000000054c514 /system/lib64/libart.so (ExecuteMterpImpl+14228)

Frames #12–#16 are the interpreter implementation itself.

#17 pc 00000000002454a0 /system/priv-app/Velvet/Velvet.apk (offset 0x1322000) (com.google.android.apps.gsa.speech.e.c.c.call+28)

Frame #17 is the Java code being interpreted. This Java method corresponds to interpreter frames #12–#16.

#18 pc 00000000002547a8 /system/lib64/libart.so (\_ZN3art11interpreterL7ExecuteEPNS\_6ThreadERKNS\_20CodeItemDataAccessorERNS\_11ShadowFrameENS\_6JValueEb.llvm.780698333+496)

#19 pc 0000000000519fd8 /system/lib64/libart.so (artQuickToInterpreterBridge+1032)

#20 pc 00000000005630fc /system/lib64/libart.so (art\_quick\_to\_interpreter\_bridge+92)

Frames #18–#20 are the VM itself, code to transition from compiled Java code to interpreted Java code.

#21 pc 00000000002ce44c /system/framework/arm64/boot.oat (offset 0xdc000) (java.util.concurrent.FutureTask.run+204)

Frame #21 is the compiled Java method that calls the Java method in #17.

#22 pc 0000000000559f88 /system/lib64/libart.so (art\_quick\_invoke\_stub+584)

#23 pc 00000000000ced40 /system/lib64/libart.so (art::ArtMethod::Invoke(art::Thread\*, unsigned int\*, unsigned int, art::JValue\*, char const\*)+200)

#24 pc 0000000000280cf0 /system/lib64/libart.so (art::interpreter::ArtInterpreterToCompiledCodeBridge(art::Thread\*, art::ArtMethod\*, art::ShadowFrame\*, unsigned short, art::JValue\*)+344)

#25 pc 000000000027acac /system/lib64/libart.so (bool art::interpreter::DoCall<false, false>(art::ArtMethod\*, art::Thread\*, art::ShadowFrame&, art::Instruction const\*, unsigned short, art::JValue\*)+948)

#26 pc 0000000000529880 /system/lib64/libart.so (MterpInvokeVirtual+584)

#27 pc 000000000054c514 /system/lib64/libart.so (ExecuteMterpImpl+14228)

Frames #22–#27 are the interpreter implementation, making a method invocation from interpreted code to a compiled method.

#28 pc 00000000003ed69e /system/priv-app/Velvet/Velvet.apk (com.google.android.apps.gsa.shared.util.concurrent.b.e.run+22)

Frame #28 is the Java code being interpreted.

#29 pc 00000000002547a8 /system/lib64/libart.so (\_ZN3art11interpreterL7ExecuteEPNS\_6ThreadERKNS\_20CodeItemDataAccessorERNS\_11ShadowFrameENS\_6JValueEb.llvm.780698333+496)

#30 pc 0000000000519fd8 /system/lib64/libart.so (artQuickToInterpreterBridge+1032)

#31 pc 00000000005630fc /system/lib64/libart.so (art\_quick\_to\_interpreter\_bridge+92)

Frames #29–#31 are another transition between compiled code and interpreted code.

#32 pc 0000000000329284 /system/framework/arm64/boot.oat (offset 0xdc000) (java.util.concurrent.ThreadPoolExecutor.runWorker+996)

#33 pc 00000000003262a0 /system/framework/arm64/boot.oat (offset 0xdc000) (java.util.concurrent.ThreadPoolExecutor$Worker.run+64)

#34 pc 00000000002037e8 /system/framework/arm64/boot.oat (offset 0xdc000) (java.lang.Thread.run+72)

Frames #32–#34 are compiled Java frames calling each other directly. In this case the native call stack is the same as the Java call stack.

#35 pc 0000000000559f88 /system/lib64/libart.so (art\_quick\_invoke\_stub+584)

#36 pc 00000000000ced40 /system/lib64/libart.so (art::ArtMethod::Invoke(art::Thread\*, unsigned int\*, unsigned int, art::JValue\*, char const\*)+200)

#37 pc 0000000000280cf0 /system/lib64/libart.so (art::interpreter::ArtInterpreterToCompiledCodeBridge(art::Thread\*, art::ArtMethod\*, art::ShadowFrame\*, unsigned short, art::JValue\*)+344)

#38 pc 000000000027acac /system/lib64/libart.so (bool art::interpreter::DoCall<false, false>(art::ArtMethod\*, art::Thread\*, art::ShadowFrame&, art::Instruction const\*, unsigned short, art::JValue\*)+948)

#39 pc 0000000000529f10 /system/lib64/libart.so (MterpInvokeSuper+1408)

#40 pc 000000000054c594 /system/lib64/libart.so (ExecuteMterpImpl+14356)

Frames #35–#40 are the interpreter itself.

#41 pc 00000000003ed8e0 /system/priv-app/Velvet/Velvet.apk (com.google.android.apps.gsa.shared.util.concurrent.b.i.run+20)

Frame #41 is the Java code being interpreted.

#42 pc 00000000002547a8 /system/lib64/libart.so (\_ZN3art11interpreterL7ExecuteEPNS\_6ThreadERKNS\_20CodeItemDataAccessorERNS\_11ShadowFrameENS\_6JValueEb.llvm.780698333+496)

#43 pc 0000000000519fd8 /system/lib64/libart.so (artQuickToInterpreterBridge+1032)

#44 pc 00000000005630fc /system/lib64/libart.so (art\_quick\_to\_interpreter\_bridge+92)

#45 pc 0000000000559f88 /system/lib64/libart.so (art\_quick\_invoke\_stub+584)

#46 pc 00000000000ced40 /system/lib64/libart.so (art::ArtMethod::Invoke(art::Thread\*, unsigned int\*, unsigned int, art::JValue\*, char const\*)+200)

#47 pc 0000000000460d18 /system/lib64/libart.so (art::(anonymous namespace)::InvokeWithArgArray(art::ScopedObjectAccessAlreadyRunnable const&, art::ArtMethod\*, art::(anonymous namespace)::ArgArray\*, art::JValue\*, char const\*)+104)

#48 pc 0000000000461de0 /system/lib64/libart.so (art::InvokeVirtualOrInterfaceWithJValues(art::ScopedObjectAccessAlreadyRunnable const&, \_jobject\*, \_jmethodID\*, jvalue\*)+424)

#49 pc 000000000048ccb0 /system/lib64/libart.so (art::Thread::CreateCallback(void\*)+1120)

Frames #42–#49 are the VM itself. This time it's the code that starts running Java on a new thread.

#50 pc 0000000000082e24 /system/lib64/libc.so (\_\_pthread\_start(void\*)+36)

#51 pc 00000000000233bc /system/lib64/libc.so (\_\_start\_thread+68)

Frames #50–#51 are how all threads should start. This is the libc new thread start code.

# **LOGGING**

1. Log Standards

Logging in Android is complex due to the mix of standards used that are combined in logcat. The main standards used are detailed below:

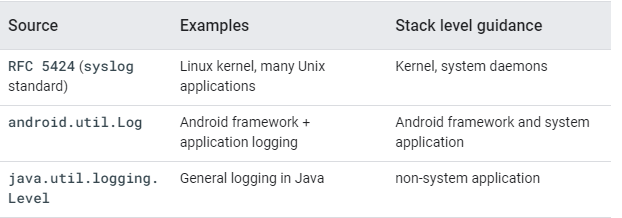


Figure 1: Log level standards.

Though each of these standards have similar level construction, they vary in granularity. Approximate equivalents across the standards are as follows:

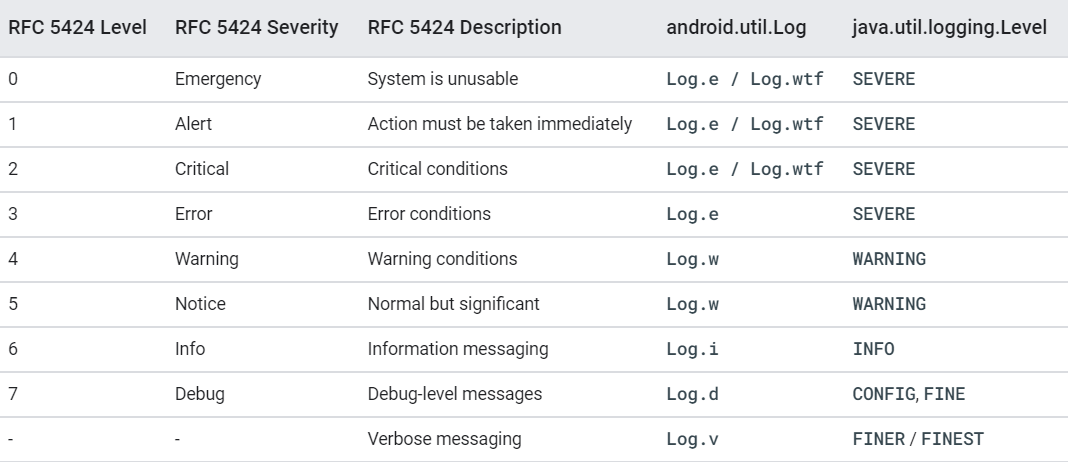


Figure 2: syslog, Android, and Java logging levels.

1. Log Level Guidelines

There are existing guidelines given for each log standard. The chosen log level follows the appropriate standard being used, like using the syslog standard for kernel development.

Log level orders, from least-to-most, are shown in the three figures below:



Figure 3: android.util.Log



Figure 4: java.util.Logging.Level.

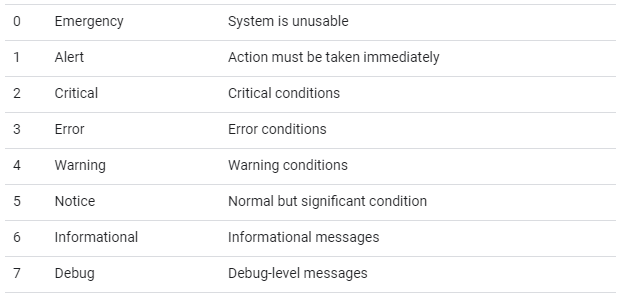


Figure 5: RFC 5424

1. Application Logging

Selective logging is performed with TAG by android.util.Log class using Log#isLoggable, as shown below:

if (Log.isLoggable("FOO\_TAG", Log.VERBOSE)) {

Log.v("FOO\_TAG", "Message for logging.");

}

Logs can be tuned at runtime to provide a select level of logging as shown below:

adb shell setprop log.tag.FOO\_TAG VERBOSE

log.tag.\* properties are reset on reboot. There are persistent variants that remain across reboots as well. See below:

adb shell setprop persist.log.tag.FOO\_TAG VERBOSE

Log#isLoggable checks leave log traces in the application code. Boolean DEBUG flags bypass log traces using compiler optimizations that are set to false, as shown below:

private final static boolean DEBUG = false;

…

If (DEBUG) {

Log.v("FOO\_TAG", "Extra debug logging.");

}

Logging can be removed on a per-APK basis via ProGuard rulesets by R8 at compile time. The following example removes everything below INFO level logging for android.util.Log:

# This allows proguard to strip isLoggable() blocks containing only <=INFO log

# code from release builds.

-assumenosideeffects class android.util.Log {

static \*\*\* i(...);

static \*\*\* d(...);

static \*\*\* v(...);

static \*\*\* isLoggable(...);

}

-maximumremovedandroidloglevel 4

This is useful for handling multiple application build types (for example, development builds vs. release builds) where the underlying code is expected to be the same, but the allowed log levels are different. An explicit policy must be set and followed for applications (particularly system applications) to decide how build types and release expectations impact log output.

1. System Logging in the JVM

There are several available classes that are available for system applications and services:

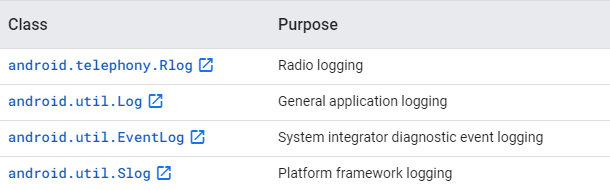


Figure 6: Available system log classes and purposes.

Though android.util.Log and android.util.Slog use the same log level standards, Slog is an @hide class usable only by the platform. The EventLog levels are mapped to the entries in the event.logtags file in /system/etc/event-log-tags.

1. Native Logging

Logging in C/C++ follows the syslog standard with syslog(2) corresponding to the Linux kernel syslog that controls the printk buffer, and syslog(3) corresponding to the general system logger. Android uses the liblog library for general system logging.

liblog provides wrappers for the sublogs groups using the following macro form:

[Sublog Buffer ID] LOG [Log Level ID]

RLOGD, for example, corresponds to [Radio log buffer ID] LOG [Debug Level]. The major liblog wrappers are as follows:

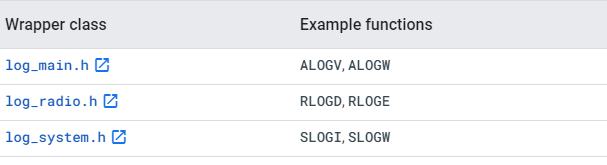


Figure 7: liblog wrappers.

Android has higher level interfaces for logging that are favoured over direct liblog usage, as seen below:

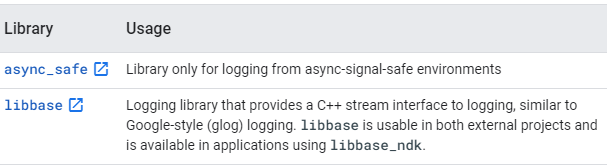


Figure 8: Higher level log Libraries.

1. Multistack Approximations

Due to differences in granularity and level intent, there are no clear or exact matchings of different logging standards. For example, the java.util.logging.Level and android.util.Log levels for error logs are not a 1:1 match:

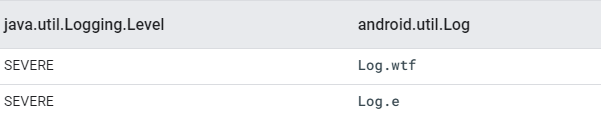


Figure 9: Error level in standard Java logging vs. Android logging.

In cases like this, use the individual standard to determine which level to apply.

During system development with multiple stack level components, follow Figure 1 to determine which standard to use per-component. For an approximate guide to tier messaging, follow Figure 2.

1. Security and Privacy

Do not log Personally Identifiable Information (PII). This includes details such as:

* Email addresses
* Telephone numbers
* Names

Similarly, certain details are considered sensitive even if not explicitly personally identifiable.

For example, though timezone info is not considered personally identifiable, it does give an indication of the approximate location of a user.

Log policy and acceptable details must be handled as part of security and privacy review before release.

## Implementing Scoped Vendor Logging

Android 11 adds a new HAL, IDumpstateDevice (version 1.1). This HAL exposes new methods to more tightly scope vendor logs that are included in standard bug reports, as well as to allow user builds to turn vendor logging on and off (the default for user builds is off). This gives OEMs more control over what gets included in particular types of bug reports.

This feature impacts OEMs if they choose to implement this optional HAL. SoCs might be impacted, depending on what the OEM chooses to expose with this HAL. There’s no expected impact to carriers.

What you include in bug reports depends on which information you find relevant for debugging, but generally more verbose is better.

1. Examples and source

There's a default implementation of the (deprecated) 1.0 version of IDumpstateDevice that shows an example of using the dumpstate util library: frameworks/native/cmds/dumpstate/DumpstateUtil.h. There's also a Cuttlefish implementation of the 1.1 HAL: device/google/cuttlefish/guest/monitoring/dumpstate\_ext/\*.

The source code is located here:

* The HAL files are under hardware/interfaces/dumpstate/1.1/.
* The dumpstate native code that controls bug report contents is under frameworks/native/cmds/dumpstate/.

1. Implementation

To implement his HAL, implement the

android.hardware.dumpstate@1.1::IDumpstateDevice HAL interface. There are many possible DumpstateMode values, but not all are likely to be supported by a single device (for example, WEAR for non-Wear OS devices).

Implementing the dumpstate HAL is optional. All new devices launching with Android 11 MUST implement IDumpstateDevice 1.1 if they implement the Dumpstate HAL. Devices that have already implemented IDumpstateDevice 1.0 prior to Android 11 should be relatively easy to upgrade to 1.1, and doing so is strongly recommended, as it greatly reduces the amount of extraneous private information included in bug reports.

This feature depends on the core dumpstate changes also included with Android 11, located under frameworks/native/cmds/dumpstate.

Implementing this HAL will likely require some SEPolicy changes to certain system properties, files, etc. to get things fully working, and will require coordination with vendors to dump all relevant information into bug reports.

1. Customization

The device user can toggle vendor logging on or off using developer settings. When it's turned off, dumpstateBoard\_1\_1 may still output minimal essential information as determined by the OEM. Turning vendor logging off makes IDumpstateDevice::dumpstateBoard add only essential information to a bug report, while turning it on includes whatever information the OEM chooses.

You can modify dumpstate.cpp (which calls the IDumpstateDevice HAL methods), for example, to increase the timeout given for dumpstateBoard to complete. However, the core logic of dumpstate.cpp should remain unchanged.

Timeouts can be any value, but they shouldn't dramatically increase the time that a bug report takes to complete. In particular, DumpstateMode::CONNECTIVITY is highly time sensitive and needs to run as fast as possible to collect all relevant modem /Wi-Fi/networking logs.

1. Validation

There's a VTS test for the IDumpstateDevice implementation, and there are functional unit tests for general BugreportManager functionality.

The recommended manual test case is frameworks/base/core/tests/bugreports/src/android/server/bugreports/BugreportManagerTest.java.

## Diagnosing Native Crashes

1. Abort

Aborts are interesting because they are deliberate. There are many different ways to abort (including calling abort(3), failing an assert(3), using one of the Android-specific fatal logging types), but all involve calling abort. A call to abort 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.

There may be an explicit "abort message" line. You should also look in the logcat output to see what this thread logged before deliberately killing itself, because unlike assert(3) or high level fatal logging facilities, abort(3) doesn't accept a message.

Current versions of Android inline the tgkill(2) system call, so their stacks are the easiest to read, with the call to abort(3) at the very top:

pid: 4637, tid: 4637, 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 0000121d r2 00000006 r3 00000008

r4 0000121d r5 0000121d r6 ffb44a1c r7 0000010c

r8 00000000 r9 00000000 r10 00000000 r11 00000000

ip ffb44c20 sp ffb44a08 lr eace2b0b pc eace2b16

backtrace:

#00 pc 0001cb16 /system/lib/libc.so (abort+57)

#01 pc 0001cd8f /system/lib/libc.so (\_\_assert2+22)

#02 pc 00001531 /system/bin/crasher (do\_action+764)

#03 pc 00002301 /system/bin/crasher (main+68)

#04 pc 0008a809 /system/lib/libc.so (\_\_libc\_init+48)

#05 pc 00001097 /system/bin/crasher (\_start\_main+38)

Older versions of Android followed a convoluted path between the original abort call (frame 4 here) and the actual sending of the signal (frame 0 here). This was especially true on 32-bit ARM, which added \_\_libc\_android\_abort (frame 3 here) to the other platforms' sequence of raise/pthread\_kill/tgkill:

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

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

1. 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) 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.

1. 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) (or any other function taking a FILE\*) and readdir(3), because code often fails to check that the fopen(3) or opendir(3) 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) 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 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.

1. 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.

1. 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 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.

1. Seccomp SIGSYS from a disallowed system call

The seccomp system (specifically seccomp-bpf) restricts access to system calls. For more information about seccomp for platform developers, see the blog post Seccomp filter in Android O. A thread that calls a restricted system call will receive a SIGSYS signal with code SYS\_SECCOMP. The system call number will be shown in the cause line, along with the architecture. It is important to note that system call numbers vary between architectures. For example, the readlinkat(2) system call is number 305 on x86 but 267 on x86-64. The call number is different again on both arm and arm64. Because system call numbers vary between architectures, it's usually easier to use the stack trace to find out which system call was disallowed rather than looking for the system call number in the headers.

pid: 11046, tid: 11046, name: crasher >>> crasher <<<

signal 31 (SIGSYS), code 1 (SYS\_SECCOMP), fault addr --------

Cause: seccomp prevented call to disallowed arm system call 99999

r0 cfda0444 r1 00000014 r2 40000000 r3 00000000

r4 00000000 r5 00000000 r6 00000000 r7 0001869f

r8 00000000 r9 00000000 sl 00000000 fp fffefa58

ip fffef898 sp fffef888 lr 00401997 pc f74f3658 cpsr 600f0010

backtrace:

#00 pc 00019658 /system/lib/libc.so (syscall+32)

#01 pc 00001993 /system/bin/crasher (do\_action+1474)

#02 pc 00002699 /system/bin/crasher (main+68)

#03 pc 0007c60d /system/lib/libc.so (\_\_libc\_init+48)

#04 pc 000011b0 /system/bin/crasher (\_start\_main+72)

You can distinguish disallowed system calls from other crashes by the presence of SYS\_SECCOMP on the signal line and the description on the cause line.

You can reproduce an instance of this type of crash using crasher seccomp.

1. Execute-only memory violation (Android 10 only)

For arm64 in Android 10 only, executable segments of binaries and libraries were mapped into memory execute-only (non-readable) as a hardening technique against code-reuse attacks. This mitigation interacted badly with other mitigations and was later removed.

Making code unreadable causes intentional and unintentional reads into memory segments marked execute-only to throw a SIGSEGV with code SEGV\_ACCERR. This might occur as a result of a bug, vulnerability, data mixed with code (such as a literal pool), or intentional memory introspection.

The compiler assumes code and data are not intermixed, but issues can arise from hand-written assembly. In many cases these can be fixed by simply moving the constants to a .data section. If code introspection is absolutely necessary on executable code sections, mprotect(2) should be called first to mark the code readable, and then again to mark it unreadable after the operation is completed.

pid: 2938, tid: 2940, name: crasher64 >>> crasher64 <<<

signal 11 (SIGSEGV), code 2 (SEGV\_ACCERR), fault addr 0x5f2ced24a8

Cause: execute-only (no-read) memory access error; likely due to data in .text.

x0 0000000000000000 x1 0000005f2cecf21f x2 0000000000000078 x3 0000000000000053

x4 0000000000000074 x5 8000000000000000 x6 ff71646772607162 x7 00000020dcf0d16c

x8 0000005f2ced24a8 x9 000000781251c55e x10 0000000000000000 x11 0000000000000000

x12 0000000000000014 x13 ffffffffffffffff x14 0000000000000002 x15 ffffffffffffffff

x16 0000005f2ced52f0 x17 00000078125c0ed8 x18 0000007810e8e000 x19 00000078119fbd50

x20 00000078125d6020 x21 00000078119fbd50 x22 00000b7a00000b7a x23 00000078119fbdd8

x24 00000078119fbd50 x25 00000078119fbd50 x26 00000078119fc018 x27 00000078128ea020

x28 00000078119fc020 x29 00000078119fbcb0

sp 00000078119fba40 lr 0000005f2ced1b94 pc 0000005f2ced1ba4

backtrace:

#00 pc 0000000000003ba4 /system/bin/crasher64 (do\_action+2348)

#01 pc 0000000000003234 /system/bin/crasher64 (thread\_callback+44)

#02 pc 00000000000e2044 /apex/com.android.runtime/lib64/bionic/libc.so (\_\_pthread\_start(void\*)+36)

#03 pc 0000000000083de0 /apex/com.android.runtime/lib64/bionic/libc.so (\_\_start\_thread+64)

You can distinguish execute-only memory violations from other crashes by the cause line.

You can reproduce an instance of this type of crash using crasher xom.

1. Error detected by fdsan

Android's fdsan file descriptor sanitizer helps catch common mistakes with file descriptors such as use-after-close and double-close. See the fdsan documentation for more details about debugging (and avoiding) this class of errors.

pid: 32315, tid: 32315, name: crasher64 >>> crasher64 <<<

signal 35 (), code -1 (SI\_QUEUE), fault addr --------

Abort message: 'attempted to close file descriptor 3, expected to be unowned, actually owned by FILE\* 0x7d8e413018'

x0 0000000000000000 x1 0000000000007e3b x2 0000000000000023 x3 0000007fe7300bb0

x4 3033313465386437 x5 3033313465386437 x6 3033313465386437 x7 3831303331346538

x8 00000000000000f0 x9 0000000000000000 x10 0000000000000059 x11 0000000000000034

x12 0000007d8ebc3a49 x13 0000007fe730077a x14 0000007fe730077a x15 0000000000000000

x16 0000007d8ec9a7b8 x17 0000007d8ec779f0 x18 0000007d8f29c000 x19 0000000000007e3b

x20 0000000000007e3b x21 0000007d8f023020 x22 0000007d8f3b58dc x23 0000000000000001

x24 0000007fe73009a0 x25 0000007fe73008e0 x26 0000007fe7300ca0 x27 0000000000000000

x28 0000000000000000 x29 0000007fe7300c90

sp 0000007fe7300860 lr 0000007d8ec2f22c pc 0000007d8ec2f250

backtrace:

#00 pc 0000000000088250 /bionic/lib64/libc.so (fdsan\_error(char const\*, ...)+384)

#01 pc 0000000000088060 /bionic/lib64/libc.so (android\_fdsan\_close\_with\_tag+632)

#02 pc 00000000000887e8 /bionic/lib64/libc.so (close+16)

#03 pc 000000000000379c /system/bin/crasher64 (do\_action+1316)

#04 pc 00000000000049c8 /system/bin/crasher64 (main+96)

#05 pc 000000000008021c /bionic/lib64/libc.so (\_start\_main)

You can distinguish this from other kinds of abort by the presence of fdsan\_error in the backtrace and the specific abort message.

You can reproduce an instance of this type of crash using crasher fdsan\_file or crasher fdsan\_dir.

1. Investigating 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, 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.

1. Reading 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 6.0 and higher, so you can see exactly which version of your code crashed. Platform binaries include a BuildId by default since Android 6.0; NDK r12 and higher 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 base 0x2000)

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 has three adjacent entries. One is readable and executable (code), one is read-only (read-only data), and one is 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]

As of Android 5.0, 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]

## Using Debuggers

This page details using LLDB or GDB for OS development. For app development, see Debug your app instead, which explains how to use the Android Studio GUI (based on LLDB).

GDB is deprecated and will be removed soon. If you're switching from GDB to LLDB, you should probably start by reading the LLDB Tutorial. If you're an expert GDB user, the GDB to LLDB command map is very helpful while transitioning.

1. Prerequisites To use a debugger:

* Set up the build environment with the usual envsetup.sh command.
* Run the same lunch command you used when building.

1. Debugging running apps or processes

To connect to a running app or native daemon, use gdbclient.py with a PID.For example, to debug the process with PID 1234, run:

gdbclient.py -p 1234

The script sets up port forwarding, starts the appropriate remote debugging stub on the device, starts the debugger on the host, configures it to find symbols, and connects it to the remote debugging stub.

1. Debugging native process startup

To debug a process as it starts, use gdbclient.py with the -r option:

gdbclient.py -r /system/bin/MY\_TEST\_APP

Then, enter continue at the debugger's prompt.

1. Debugging app startup

Sometimes you want to debug an app as it starts, such as when there's a crash and you want to step through code to see what happened before the crash. Attaching works in some cases, but in other cases is impossible because the app crashes before you can attach. The logwrapper approach (used for strace) doesn't always work because the app might not have permissions to open a port, and gdbserver inherits that restriction.

To debug app startup, use the developer options in Settings to instruct the app to wait for a Java debugger to attach:

1. Go to Settings > Developer options > Select debug app and choose your app from the list, then click Wait for debugger.
2. Start the app, either from the launcher or by using the command line to run:

adb shell am start -a android.intent.action.MAIN -n APP\_NAME/.APP\_ACTIVITY

1. Wait for the app to load and a dialog to appear telling you the app is waiting for a debugger.
2. Attach gdbserver/gdbclient normally, set breakpoints, then continue the process.

To let the app run, attach a Java Debug Wire Protocol (JDWP) debugger such as Java Debugger (jdb):

adb forward tcp:12345 jdwp:XXX # (Where XXX is the PID

of the debugged process.)

jdb -attach localhost:12345

1. Debugging apps or processes that crash

If you want debuggerd to suspend crashed processes so that you can attach a debugger, set the appropriate property:

* After Android 11

adb shell setprop debug.debuggerd.wait\_for\_debugger true

* Android 11 and lower

adb shell setprop debug.debuggerd.wait\_for\_gdb true

* Android 6.0 Marshmallow and lower

adb shell setprop debug.db.uid 999999

At the end of the usual crash output, debuggerd provides copy and paste instructions in logcat showing how to connect the debugger to the crashed process.

1. Debugging without symbols

For 32-bit ARM, if you don’t have symbols, gdb can't determine which instruction set it's disassembling (ARM or Thumb). To specify the instruction set chosen as the default when symbol information is missing, set the following property:

set arm fallback-mode arm # or thumb

1. Debugging with VS Code

LLDB supports debugging platform code on Visual Studio Code. You can use the VS Code debugger frontend instead of the LLDB CLI interface to control and debug native code running on devices.

Before using VS Code for debugging, install the CodeLLDB extension.

To debug code using VS Code:

1. Ensure that all build artifacts (such as symbols) required to run gdbclient.py or lldbclient.py are present.
2. Run the following command:

lldbclient.py -p pid | -n proc-name | -r ... --setup-forwarding vscode ANY\_OTHER\_FLAGS

This prints a JSON object and lldbclient.py continues running. This is expected; don't kill the lldbclient.py program.

1. In the debugging tab in VS Code, select add configuration, then select LLDB: Custom Launch. This opens a launch.json file and adds a new JSON object to a list.
2. Delete the newly added debugger configuration.
3. Copy the JSON object printed by lldbclient.py and paste it into the object you just deleted. Save the changes.
4. To reload the window to refresh the debugger list, press Ctrl+Shift+P and type reload window.
5. Select the new debugger configuration and press run. The debugger should connect after 10 to 30 seconds.
6. When you're done debugging, go to the terminal running lldbclient.py and press Enter to end the lldbclient.py program.

**QUESTIONS:**

1. Explain the advantages of the computer system.

**Answer:**

* 1. **Computers can do the same task repetitively with the same accuracy.**
  2. **Computers do not get tired or bored.**
  3. **Computers can take up routine tasks while releasing human resources for more intelligent functions.**

1. Explain the disadvantages of the computer system.

**Answer:**

* 1. **Computers have no intelligence; they follow the instructions blindly without considering the outcome.**
  2. **Regular electric supply is necessary to make computers work, which could prove difficult everywhere especially in developing nations.**

1. Classify the size of the computer from the smallest to supercomputer.

**Answer:**

* 1. **Desktop.**
  2. **Laptop.**
  3. **Tablet.**
  4. **Server.**
  5. **Mainframe.**
  6. **Supercomputer.**

1. Explain the Important features of mainframes:

**Answer:**

* 1. **Big in size.**
  2. **Hundreds of times faster than servers, typically hundred megabytes per second.**
  3. **Very expensive.**
  4. **Use proprietary OS provided by the manufacturers.**
  5. **In-built hardware, software and firmware security features.**

1. Describe the most common uses of supercomputers:

**Answer:**

* 1. **Molecular mapping and research.**
  2. **Weather forecasting.**
  3. **Environmental research.**
  4. **Oil and gas exploration.**

1. Convert 27FB16 to decimal base number,

**Answer:**

**1023410.**

1. Convert 7268 to decimal base number.

**Answer:**

**47010.**

1. Convert 110102 to decimal base number.

**Answer:**

**2610.**

1. Convert Hexadecimal number A to binary.

**Answer:**

**10102**

1. Convert ASCII code character A to binary.

**Answer:**

**0100 00012.**

**REFERENCE:**

1. https://source.android.com/devices/tech/debug/understanding-logging
2. https://source.android.com/devices/tech/debug/scoped-vendor-logging
3. https://source.android.com/devices/tech/debug/native-crash
4. https://source.android.com/devices/tech/debug/gdb