Debugging native platform code using LLDB and GDB

Chris Simmonds, 2net

Overview

- Native debugging
- LLDB
- GDB

Native debugging platform code

- Native code = compiled code, written in C/C++
- Platform code = code build using the AOSP build system
- Examples:
  - Command line programs
  - Daemons
  - Libraries

LLDB and GDB

- A tale of two debuggers
- LLDB
  - The LLVM (Clang) debugger
  - Supported since Q/10
- GDB
  - The GNU debugger
  - Deprecated since S/12

Native code and symbol tables

- Debugging native code requires executables built with debug information
- There are debug executables for all AOSP executables in `$OUT/symbols`

For example, here is a command line helloworld program:

```
$ cd $OUT
$ file vendor/bin/helloworld-bp
vendor/bin/helloworld-bp: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically
linked, [...]  stripped

$ file symbols/vendor/bin/helloworld-bp
symbols/vendor/bin/helloworld-bp: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
dynamically linked, [...] with debug_info, not stripped
```

gdbclient.py and lldbclient.py

`development/scripts/gdbclient.py`
`development/scripts/lldbclient.py`

- They make it easy to launch the debug server on the target and connect to a command line session
- Since R/11, `gdbclient.py` works for both GDB and LLDB, LLDB is the default
- `lldbclient.py` is just symlink to `gdbclient.py`

For GDB:

```
$ gdbclient.py --no-lldb
```

Unfriendly message

```
$ gdbclient.py -r /vendor/bin/helloworld-bp
gdb is deprecated in favor of lldb. If you can't use lldb, please set --no-lldb and file a
bug asap.
```

Debugging hellowrld

- Preparing
  - Need to cd to `$ANDROID_BUILD_TOP`
  - Need to start ADB as root

The next few slides show simple session, using LLDB to run a command line program (helloworld)

```
$ croot
$ adb root
```

Debugging hellowrld

- Use lldbclient.py to start the debug session:

```
$ lldbclient.py  -r /vendor/bin/helloworld-bp
Process 1802 stopped
* thread #1, name = 'helloworld-bp', stop reason = signal SIGSTOP
    frame #0: 0x00007ffff7f175c0 linker64`__dl__start at begin.S:35
   32     // Force unwinds to end in this function.
   33     .cfi_undefined %rip
   34
-> 35     mov %rsp, %rdi
   36     call __linker_init
   37
   38     /* linker init returns (%rax) the _entry address in the main image */
```

This is the C++ runtime start code. Nothing interesting here

Debugging hellowrld

- Set a breakpoint on main() and continue (c) to the break

```
(lldb) b main
Breakpoint 1: where = helloworld-bp`main + 1 at helloworld.c:5:2, address = 0x0000555555555491
(lldb) c
Process 1802 resuming
Process 1802 stopped
* thread #1, name = 'helloworld-bp', stop reason = breakpoint 1.1
    frame #0: 0x0000555555555491 helloworld-bp`main at helloworld.c:5:2
   2
   3    int main (void)
   4    {
-> 5        printf("Hello, world!\n");
   6        return 0;
   7    }
```

Debugging hellowrld

- Step to the next line (n)

```
(lldb) n
Hello, world!
Process 1802 stopped
* thread #1, name = 'helloworld-bp', stop reason = step over
    frame #0: 0x000055555555549d helloworld-bp`main at helloworld.c:6:2
   3    int main (void)
   4    {
   5        printf("Hello, world!\n");
-> 6        return 0;
   7    }
```

Debugging hellowrld

- Continue execution: the program ends

```
(lldb) c
Process 1802 resuming
Process 1802 exited with status = 0 (0x00000000)
(lldb) q
$
```

Debugging hellowrld

- This error message means that you are not running adbd as root

```
$ lldbclient.py  -r /vendor/bin/helloworld-bp
[...]
(lldb) gdb-remote 5039
error: failed to get reply to handshake packet
```

Getting to know LLDB commands

- LLDB Tutorial `https://lldb.llvm.org/use/tutorial.html`
- GDB to LLDB command map `https://lldb.llvm.org/use/map.html`

Debugging the lighs HAL

- This is an example of attaching to a running daemon
- Uses the -p [PID] option to attach to a running process

First, find the PID

```
 ps -A | grep lights
nobody          435     1   18876   4944 binder_wait_for_work 0 S android.hardware.lights-service.marvin
```

Then attach

```
$ lldbclient.py -p 435
[...]
(lldb) gdb-remote 5039
Process 435 stopped
* thread #1, name = 'android.hardwar', stop reason = signal SIGSTOP
    frame #0: 0xf21af509 [vdso]
->  0xf21af509: popl   %ebp
    0xf21af50a: popl   %edx
    0xf21af50b: popl   %ecx
    0xf21af50c: retl
```

Debugging the lighs HAL

Use bt (backtrack) to find out where we are

```
(lldb) bt
* thread #1, name = 'android.hardwar', stop reason = signal SIGSTOP
  * frame #0: 0xf21af509 [vdso]
    frame #1: 0xf121a2bc libbinder.so`android::IPCThreadState::talkWithDriver(this=0xf0e41220, doReceive=<unavailable>) at IPC
    frame #2: 0xf121a66b libbinder.so`android::IPCThreadState::getAndExecuteCommand(this=0xf0e41220) at IPCThreadState.cpp:524
    frame #3: 0xf121b252 libbinder.so`android::IPCThreadState::joinThreadPool(this=0xf0e41220, isMain=<unavailable>) at IPCThr
    frame #4: 0xf1d5b976 libbinder_ndk.so`::ABinderProcess_joinThreadPool() at process.cpp:35:29
    frame #5: 0x5e6d71d2 android.hardware.lights-service.marvin`main at main.cpp:33:5
    frame #6: 0xf1473d6c libc.so`::__libc_init(raw_args=0xff95a7b0, onexit=0x00000000, slingshot=(android.hardware.lights-serv
    frame #7: 0x5e6d66bb android.hardware.lights-service.marvin`_start_main(raw_args=0xff95a7b0) at crtbegin.c:45:3
    frame #8: 0x5e6d6670 android.hardware.lights-service.marvin`_start + 16
(lldb)
```

Debugging the lighs HAL

Find the full reference for function setLightState

```
(lldb) image lookup -r -s setLightState
1 symbols match the regular expression 'setLightState' in /home/chris/aosp/out/target/product/marvin/symbols/vendor/bin/hw/android.har
        Address: android.hardware.lights-service.marvin[0x00002780] (android.hardware.lights-service.marvin.PT_LOAD[1]..text + 288)
        Summary: android.hardware.lights-service.marvin`aidl::android::hardware::light::Lights::setLightState(int, aidl::android::hard
1 symbols match the regular expression 'setLightState' in /home/chris/aosp/out/target/product/marvin/symbols/vendor/bin/hw/android.har
        Address: android.hardware.lights-service.marvin[0x00002780] (android.hardware.lights-service.marvin.PT_LOAD[1]..text + 288)
        Summary: android.hardware.lights-service.marvin`aidl::android::hardware::light::Lights::setLightState(int, aidl::android::hard
2 symbols match the regular expression 'setLightState' in /home/chris/aosp/out/target/product/marvin/symbols/apex/com.android.vndk.v31
        Address: android.hardware.light-V1-ndk_platform.so[0x00005e20] (android.hardware.light-V1-ndk_platform.so.PT_LOAD[1]..text + 7
        Summary: android.hardware.light-V1-ndk_platform.so`aidl::android::hardware::light::ILightsDefault::setLightState(int, aidl::an
        Summary: android.hardware.light-V1-ndk_platform.so`aidl::android::hardware::light::BpLights::setLightState(int, aidl::android:
```

Debugging the lighs HAL

Set a breakpoint and continue

```
(lldb) b Lights::setLightState
(lldb) c
Process 435 resuming
```

Wait until the breakpoint is hit

```
Process 435 stopped
* thread #1, name = 'android.hardwar', stop reason = breakpoint 1.1
    frame #0: 0x5e6d67a5 android.hardware.lights-service.marvin`aidl::android::hardware::light::Lights::setLightState(this=<unavailabl
   24   namespace light {
   25
   26   ndk::ScopedAStatus Lights::setLightState(int id, const HwLightState& state) {
-> 27       LOG(INFO) << "Lights setting state for id=" << id << " to color " << std::hex << state.color;
   28       if (id < 0 || id > 2) {
   29           return ndk::ScopedAStatus::fromExceptionCode(EX_UNSUPPORTED_OPERATION);
   30       } else {
```

Debugging platform JNI libraries

- Much of the Android framework is implemented in the Java program `system_server`
- (We covered debugging Java platform code in the May Meetup: `https://2net.co.uk/slides/aosp-aaos-meetup/2022-may-debug.pdf`)
- But, how to debug the JNI libraries used by Java code such as `system_server`?

Debugging platform JNI libraries

Find the PID of System Server

```
1|marvin:/ $ ps -A | grep system_server
system          584   357 2436768 347720 0                   0 S system_server
```

Attach to that PID

```
$ lldbclient.py -p 584
```

Set breakpoints on the functions you want to debug (e.g. `setLight_native`)

```
(lldb) b setLight_native
Breakpoint 1: where = libandroid_servers.so`android::setLight_native(_JNIEnv*, _jobject*, int, int, int, int, int, int) + 34
at com_android_server_lights_LightsService.cpp:136:9, address = 0xbe46a3f2
(lldb) c
Process 584 resuming
```

(*) attaching to `system_server` take a while because it has so many threads that each have to be stopped

Debugging with VS Code

- install the CodeLLDB extension:
  https://marketplace.visualstudio.com/items?itemName=vadimcn.vscode-lldb
- ```
  $ lldbclient.py --setup-forwarding vscode-lldb -r program_to_run
  ```
- 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
- Delete the new debugger configuration, then paste in copy the JSON object printed by lldbclient.py
- Press Ctrl+Shift+P and type reload window
- Select the new debugger configuration and press run. The debugger should connect after 10 to 30 seconds.
  ```