Red Hat

Support for mini-debuginfo in LLDB

How to read the .gnu_debugdata section

Konrad Kleine

February 2, 2020

Red Hat

About me

- Senior Software Engineer at Red Hat
- LLDB, C/C++, ELF, DWARF since mid 2019
- Before worked on Go and OpenShift since mid 2016

Reach out

- https://www.linkedin.com/in/konradkleine
- https://github.com/kwk/talks/
- https://developers.redhat.com/blog/author/kkleine/

The Plan In Hindsight

⌖ Overall goal and first steps

Make LLDB a better debugger for Fedora and RHEL binaries when no debug symbols have been installed.

Tackle the problem

- Vague knowledge about `.gnu_debugdata` nor ELF or alike
- Not deal with how `.gnu_debugdata` is produced, **just consume it**
    - For integration with LLDB's tests we eventually have to produce it
- Take existing Fedora binary (`/usr/bin/zip`)
    - Identify a symbol/function
        - not immediately visible in the main binary's `.dynsym` but nested within in `.gnu_debugdata`
    - Set breakpoint on that function with GDB to see if it can find it and hit it when executing

Extract .gnu_debugdata section to zip.gdd.xz

```
~$ cp /usr/bin/zip .
~$ objcopy --dump-section .gnu_debugdata=zip.gdd.xz zip
~$ file zip.gdd.xz
zip.gdd.xz: XZ compressed data
```

Notes

- objcopy creates a temporary file next to the executable
  - /usr/bin requires root
  - hence, copy binary over to user's home for inspection
- eu-readelf -Ws --elf-section /usr/bin/zip to directly inspect symbols within .gnu_debugdata but we eventually need to implement our own extraction within LLDB anyways
- .xz file format described here: https://tukaani.org/xz/xz-file-format.txt

Decompress `zip.gdd.xz` to `zip.gdd`

```
~$ xz --decompress --keep zip.gdd.xz

~$ file zip.gdd

zip.gdd: ELF 64-bit LSB executable, \
x86-64, \
version 1 (SYSV), \
dynamically linked, \
interpreter *empty*, \
BuildID[sha1]=de743a8b79536e16856de1cef558ab6700675302, \
for GNU/Linux 3.2.0, not stripped
```

Notice
A section inside the main binary contains a compressed ELF file on it's own!

Identify symbol in `zip.gdd` but not in main binary

```
~$ eu-readelf -s zip.gdd

Symbol table [28] '.symtab' contains 202 entries:
 82 local symbols  String table: [29] '.strtab'
  Num:            Value   Size Type    Bind   Vis          Ndx Name
    0: 0000000000000000      0 NOTYPE  LOCAL  DEFAULT    UNDEF
    1: 0000000000408db0    494 FUNC    LOCAL  DEFAULT       15 freeup
    2: 0000000000408fa0   1015 FUNC    LOCAL  DEFAULT       15 DisplayRunningStats
    3: 00000000004093a0    128 FUNC    LOCAL  DEFAULT       15 help
[...]
```

This `help` symbol looks promising[1]. Double check that it's **not** in the main binary's `.dynsym`:

```
~$ eu-readelf --symbols /usr/bin/zip | grep help
~$
```

[1] Promising as in: we may be able to trigger it with `/usr/bin/zip --help`.

Set and hit breakpoint on `help` with GDB 8.3[2]

```
~$ gdb --nx --args /usr/bin/zip --help
Reading symbols from /usr/bin/zip...
Reading symbols from .gnu_debugdata for /usr/bin/zip...
(No debugging symbols found in .gnu_debugdata for /usr/bin/zip)
Missing separate debuginfos, use: dnf debuginfo-install zip-3.0-25.fc31.x86_64
(gdb) b help
Breakpoint 1 at 0x4093a0
(gdb) r
Starting program: /usr/bin/zip --help

Breakpoint 1, 0x00000000004093a0 in help ()
(gdb)
```

Success and two things to note:

1. Symbols read from `.gnu_debugdata`
2. No debug symbols installed for zip

[2] GDB 8.3 is what ships with Fedora 31

Set and hit breakpoint on help with LLDB 9.0.0[3]

```
~$ lldb -x /usr/bin/zip -- --help
(lldb) target create "/usr/bin/zip"
Current executable set to '/usr/bin/zip' (x86_64).
(lldb) settings set -- target.run-args  "--help"
(lldb) b help
Breakpoint 1: no locations (pending).
WARNING:  Unable to resolve breakpoint to any actual locations.
(lldb) r
[... OUTPUT OF: /usr/bin/zip --help ...]
Process 83336 exited with status = 0 (0x00000000)
(lldb)
```

Error :/
Evidently, LLDB *had* no idea of how to make use of `.gnu_debugdata`, yet.

[3] LLDB 9.0.0 is what ships with Fedora 31

Hack LLDB

Modify LLDB’s CMake build system to get LZMA in

Lempel–**Z**iv–**M**arkov chain **A**lgorithm used to decompress .xz files

lldb/cmake/modules/LLDBConfig.cmake:

```
include(CMakeDependentOption)
//...
find_package(LibLZMA)
cmake_dependent_option(LLDB_ENABLE_LZMA
  "Support LZMA compression"
  ON "LIBLZMA_FOUND" OFF)
if (LLDB_ENABLE_LZMA)
  include_directories(${LIBLZMA_INCLUDE_DIRS})
endif()
llvm_canonicalize_cmake_booleans(LLDB_ENABLE_LZMA)
```

Module for finding LZMA and dependent option macro come with CMake:

- https://gitlab.kitware.com/cmake/cmake/blob/master/Modules/FindLibLZMA.cmake
- https://gitlab.kitware.com/cmake/cmake/blob/master/Modules/CMakeDependentOption.cmake

Implement reusable LZMA decompression and helpers

From **ldb/Host/LZMA.h**:

```
// returns true if LZMA is available so no ifdef's needed in consuming code
bool isAvailable();

// 1. decodes the LZMA footer: lzma_stream_footer_decode(...)
// 2. reads and decodes the LZMA index: lzma_index_buffer_decode(...FOOTER...)
// 3. returns size of uncompressed xz-file: lzma_index_uncompressed_size(...INDEX...)
llvm::Expected<uint64_t>
getUncompressedSize(llvm::ArrayRef<uint8_t> InputBuffer);

// resizes Uncompressed to result of getUncompressedSize(...)
// and decodes Input into Uncompressed: lzma_stream_buffer_decode(...Input...)
llvm::Error uncompress(llvm::ArrayRef<uint8_t> InputBuffer,
                       llvm::SmallVectorImpl<uint8_t> &Uncompressed);
```

Read the `.gnu_debugdata` section

If there's a `.gnu_debugdata` section, we'll try to read the `.symtab` that's embedded in there and replace the one in the original object file (if any). If there's none in the orignal object file, we add it to it.

lldb/source/Plugins/ObjectFile/ELF/ObjectFileELF.cpp:

```
if (auto gdd_obj_file = GetGnuDebugDataObjectFile()) {
  if (auto gdd_objfile_section_list = gdd_obj_file->GetSectionList()) {
    if (SectionSP symtab_section_sp =
            gdd_objfile_section_list->FindSectionByType(
                eSectionTypeELFSymbolTable, true)) {
      SectionSP module_section_sp = unified_section_list.FindSectionByType(
          eSectionTypeELFSymbolTable, true);
      if (module_section_sp)
        unified_section_list.ReplaceSection(module_section_sp->GetID(),
                                            symtab_section_sp);
      else
        unified_section_list.AddSection(symtab_section_sp);
    }
  }
}
```

⚠ Let's talk `.symtab`

Symtab

- normally, `.dynsym` is a subset of `.symtab`.
- but `.gnu_debugdata`'s embedded `.symtab` has `.dynsym` symbols stripped[4]:

```
# Keep all the function symbols not already in the dynamic symbol
# table.
comm -13 dynsyms funcsyms > keep_symbols
```

Implications for LLDB

- thus, LLDB needs to load **both**
  - before it either loaded `.symtab` or `.dynsym`

```
if (!symtab) {
  // [...]
  symtab =
      section_list->FindSectionByType(eSectionTypeELFDynamicSymbols, true)
          .get();
}
```

[4] https://sourceware.org/gdb/current/onlinedocs/gdb/MiniDebugInfo.html

Changes to LLDB

LLDB now parses the `.dynsym` symbol table when no `.symtab` was found or when `.gnu_debugdata` was found.

`Symtab *ObjectFileELF::GetSymtab()`:

```
// The symtab section is non-allocable and can be stripped, while the
// .dynsym section which should always be always be there. To support the
// minidebuginfo case we parse .dynsym when there's a .gnu_debuginfo
// section, nomatter if .symtab was already parsed or not. This is because
// minidebuginfo normally removes the .symtab symbols which have their
// matching .dynsym counterparts.
if (!symtab ||
    GetSectionList()->FindSectionByName(ConstString(".gnu_debugdata"))) {
  Section *dynsym =
      section_list->FindSectionByType(eSectionTypeELFDynamicSymbols, true)
          .get();
  if (dynsym) {
    if (!m_symtab_up)
      m_symtab_up.reset(new Symtab(dynsym->GetObjectFile()));
    symbol_id += ParseSymbolTable(m_symtab_up.get(), symbol_id, dynsym);
  }
}
```

☑ Show that LLDB can now find `help` symbol

```
$ lldb -x /usr/bin/zip -- --help
(lldb) target create "/usr/bin/zip"
Current executable set to '/usr/bin/zip' (x86_64).
(lldb) settings set -- target.run-args  "--help"
(lldb) b help
Breakpoint 1: where = zip`help, address = 0x00000000004093a0
(lldb) r
Process 277525 launched: '/usr/bin/zip' (x86_64)
Process 277525 stopped
* thread #1, name = 'zip', stop reason = breakpoint 1.1
    frame #0: 0x00000000004093a0 zip`help
zip`help:
->  0x4093a0 <+0>:  pushq  %r12
    0x4093a2 <+2>:  movq   0x2af6f(%rip), %rsi       ;  + 4056
    0x4093a9 <+9>:  movl   $0x1, %edi
    0x4093ae <+14>: xorl   %eax, %eax
(lldb)
```

☑ Testing in LLVM

lit: LLVM-Integrated-Tester[6]

- lit file can
  - be test and input all at once (example follows)
  - contain `RUN`, `CHECK`, `REQUIRES` comments (simplified)
  - typically makes use of a tool called `FileCheck`[5]
- lit makes no assumption about the type of file (e.g. `*.yaml`, `*.c`, etc.)
- lit substitutes a bunch of variables in comments:

Macro	Substitution
%s	source path (path to the file currently being run)
%t	temporary file name unique to the test

[5] https://llvm.org/docs/CommandGuide/FileCheck.html

[6] https://llvm.org/docs/CommandGuide/lit.html

Example test file in Shell test suite

lldb/test/Shell/Breakpoint/example.c:

```
// REQUIRES: system-linux, lzma, xz
// RUN: gcc -g -o %t %s
// RUN: %t 1 2 3 4 | FileCheck --dump-input=always --color %s
#include <stdio.h>
int main(int argc, char* argv[]) {
  // CHECK: Number of {{.*}}: 5
  printf("Number of arguments: %d\n", argc);
  if (argc > 1) {
    // CHECK-NEXT: more than the program path
    printf("more than the program path\n");
  }
  return 0;
}
```

```
~/llvm-project$ llvm-lit -av lldb/test/Shell/Breakpoint/example.c
```

Don’t invoke compiler directly, look in other tests!

Lit example output (slightly modified)

```
-- Testing: 1 tests, 1 workers --
PASS: lldb-shell :: Breakpoint/example.c (1 of 1)
Script:
--
: 'RUN: at line 2';   gcc -g -o ~/llvm-build/tools/lldb/test/Breakpoint/Output/example.c.tmp \
                      ~/llvm/lldb/test/Shell/Breakpoint/example.c
: 'RUN: at line 3';   ~/llvm-build/tools/lldb/test/Breakpoint/Output/example.c.tmp 1 2 3 4 \
                      | ~/llvm-build/bin/FileCheck \
                        --dump-input=always --color ~/llvm/lldb/test/Shell/Breakpoint/example.c
--
Exit Code: 0
Command Output (stderr):
--
Input file: <stdin>
Check file: ~/llvm/lldb/test/Shell/Breakpoint/example.c

Full input was:
<<<<<<
   1: Number of arguments: 5
   2: more than the program path
>>>>>>

Testing Time: 0.98s
  Expected Passes    : 1
```

Modify LLDB’s integrated tester config

1. check if LZMA was compiled with LLVM
2. check if the `xz` executable was found on the system

lldb/test/Shell/lit.site.cfg.py.in[7]:

```
config.lldb_enable_lzma = @LLDB_ENABLE_LZMA@
```

lldb/test/Shell/lit.cfg.py:

```
#...
if config.lldb_enable_lzma:
    config.available_features.add('lzma')

if find_executable('xz') != None:
    config.available_features.add('xz')
# ...
```

Used here when requiring features for a test:

```
// REQUIRES: system-linux, lzma, xz
```

[7] For `LLDB_ENABLE_LZMA` see the changes to CMake

Thank you!

You can find this talk at https://github.com/kwk/talks/

Sources or recommended reads

https://sourceware.org/gdb/current/onlinedocs/gdb/MiniDebugInfo.html#MiniDebugInfo

https://github.com/rpm-software-management/rpm/blob/7cc9eb84a3b2baa0109be599572d78870e0
scripts/find-debuginfo.sh#L261

https://gnu.wildebeest.org/blog/mjw/2016/02/02/where-are-your-

Tests in LLDB (1848) and Clang (11686) by suites

Clang has 6x more tests than LLDB

```
$ ~/llvm-builds/relwithdebinfo/bin/llvm-lit --show-suites ~/llvm/lldb/test
-- Test Suites --
  lldb-api - 784 tests
    Source Root: /home/kkleine/llvm/lldb/packages/Python/lldbsuite/test
    Exec Root  : /home/kkleine/llvm/lldb/packages/Python/lldbsuite/test
  lldb-shell - 295 tests
    Source Root: /home/kkleine/llvm/lldb/test/Shell
    Exec Root  : /home/kkleine/llvm-builds/relwithdebinfo/tools/lldb/test
    Available Features : asserts dbregs-set lld lua lzma native native-cpu-avx
    native-cpu-sse python shell system-linux target-x86_64 x86 x86_64-linux xz zlib
  lldb-unit - 769 tests
    Source Root: /home/kkleine/llvm-builds/relwithdebinfo/tools/lldb/unittests
    Exec Root  : /home/kkleine/llvm-builds/relwithdebinfo/tools/lldb/unittests
    Available Features : shell system-linux target-x86_64 x86_64-linux

$ ~/llvm-builds/relwithdebinfo/bin/llvm-lit --show-suites ~/llvm/clang/test -v
-- Test Suites --
  Clang - 11686 tests
    Source Root: /home/kkleine/llvm/clang/test
    Exec Root  : /home/kkleine/llvm-builds/relwithdebinfo/tools/clang/test
    Available Features : LP64 ansi-escape-sequences asserts backtrace
    can-remove-opened-file clang-driver crash-recovery dev-fd-fs enable_shared
    libgcc native plugins shell staticanalyzer system-linux target-x86_64 thread_support
    utf8-capable-terminal x86-registered-target x86_64-linux xmllint z3 zlib
```