Red Hat
Platform
Tools Team

Support for mini-debuginfo in LLDB

How to read the .gnu_debugdata section

Konrad Kleine

February 2, 2020

Red Hat

About me

Konrad Kleine

- Red Hat
- LLDB, C/C++, ELF, DWARF since 2019
- Before worked OpenShift since 2016

Reach out

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

Overall goal

Improve LLDB for Fedora and RHEL binaries

- when no debug symbols installed
  - backtraces only show addresses
  - runtime symbols stored in special location

Approach

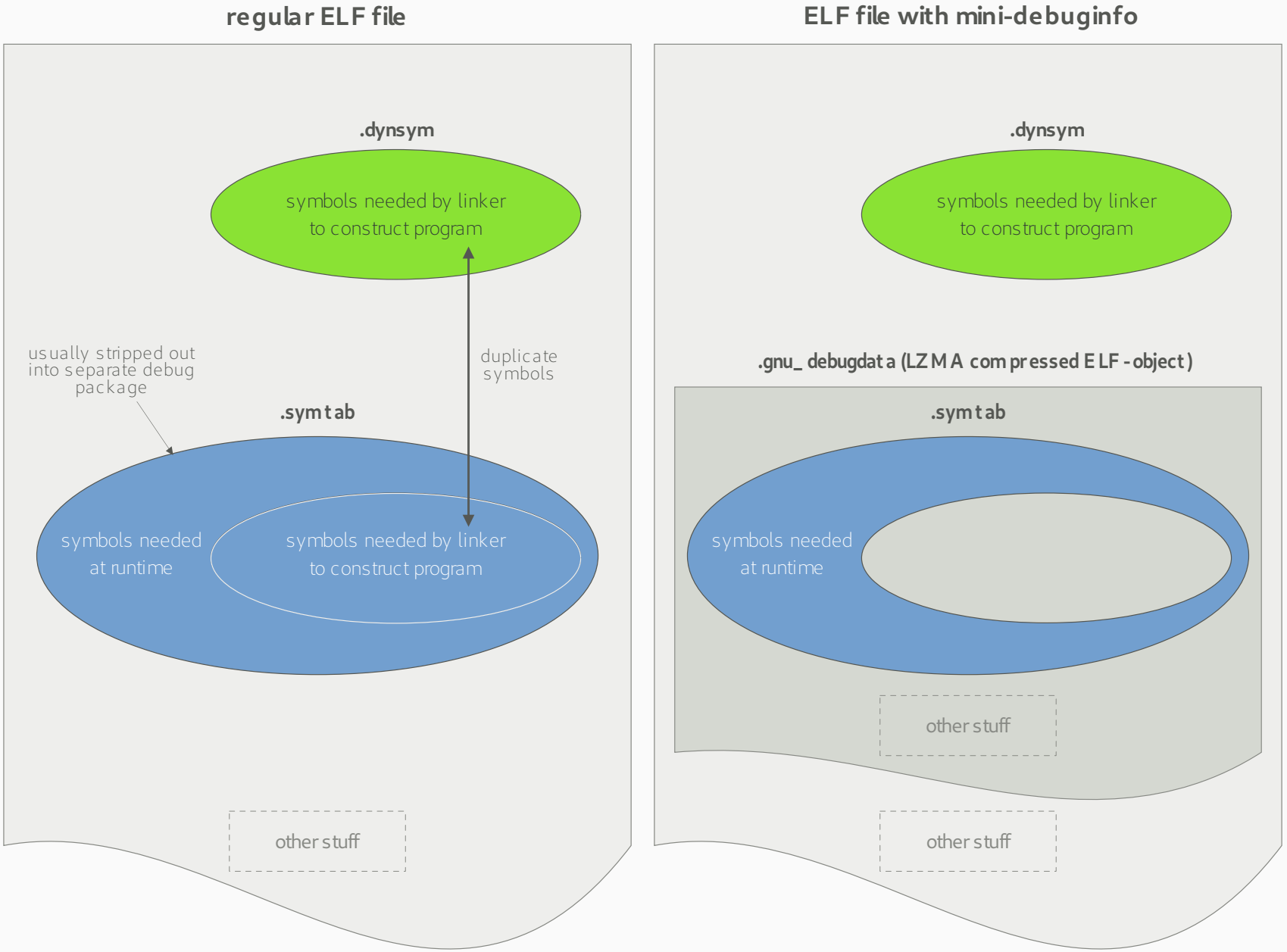
- make LLDB understand mini-debuginfo
  - that's where runtime symbols are stored

💡 Why was mini-debuginfo invented and how?

- without installing debug infos
  - be able to generate a backtrace for crashes with ABRT[1]
  - ~~have symbol table (`.symtab`)~~
  - ~~have line information (`.debug_line`)~~
  - ➔ *more than two sections make up an ELF file* ☺
- eventually only one relevant section
  - stripped `.symtab`
  - rest was too big
  - ELF format remained
  - **no replacement** for separate full debug info
  - **not related** to DWARF
    - *just symbol tables*

[1] Automatic Bug Reporting Tool

Symbol tables in an ELF file



Where and since when is mini-debuginfo being used?

- On by default since Fedora 18 (2013, Release Notes 4.2.4.1.)
- Red Hat Enterprise Linux (RHEL) since 7

Approach

Not focus on backtraces

- but make LLDB see mini-debuginfo
  - symbol awareness through
    - set and hit breakpoint
    - dump symbols (`(lldb) image dump symtab`)

Take existing Fedora binary (`/usr/bin/zip`)

- identify a symbol/function
- shootout: GDB vs. LLDB
- hurdles:
  - not from `.dynsym`
  - from within `.gnu_debugdata`

Extract + decompress .gnu_debugdata from /usr/bin/zip

```
# Dump section
~$ objcopy --dump-section .gnu_debugdata=zip.gdd.xz zip

# Determine file type of section
~$ file zip.gdd.xz
zip.gdd.xz: XZ compressed data

# Decompress section
~$ xz --decompress --keep zip.gdd.xz

# Determine file type of decompressed section
~$ file zip.gdd
zip.gdd: ELF 64-bit LSB executable, x86-64, version 1 [...]
```

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

```
# Show symbols
~$ 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
[...]
```

`help` looks promising[2].

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

[2] 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[3]

```
~$ 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

[3] GDB 8.3 is what ships with Fedora 31

Set and hit breakpoint on `help` with LLDB 9.0.0[4]

```
~$ 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)
```

♫ *If you're ☹ and you know it . . .* 

[4] LLDB 9.0.0 is what ships with Fedora 31

⚠ Let's talk `.symtab`

Symtab

- normally, `.dynsym` is subset
- **but** for mini-debuginfo `.dynsym` symbols are stripped[5]

Implications for LLDB (and other tools)

- parse `.dynsym` when
  - no `.symtab` found **or**
  - mini-debuginfo present and smuggled in

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

☑ 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)
```

♡ shipping with LLVM 10

Ready to ship?

⍰ What tests exists for mini-debuginfo?

- 🔍 find symbol from `.gnu_debugdata`
- ⚠ when mini-debuginfo w/o LZMA support
- ❗ when decompressing corrupted xz
- ⚙ full example with compiled and modified code in accordance to gdb's documentation

fell asleep yet?

</> Example test file in Shell test suite

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

```
1  // REQUIRES: system-linux, lzma, xz
2  // RUN: gcc -g -o %t %s
3  // RUN: %t 1 2 3 4 | FileCheck %s
4
5  #include <stdio.h>
6  int main(int argc, char* argv[]) {
7
8    // CHECK: Number of {{.*}}: 5
9    printf("Number of arguments: %d\n", argc);
10
11   return 0;
12 }
```

- features added: `lzma`, `xz`
  - just some CMake canonisation and Python config

```
1  if config.lldb_enable_lzma:
2    config.available_features.add('lzma')
3  if find_executable('xz') != None:
4    config.available_features.add('xz')
```

You might wonder...

What was the hardest part?

- ☺ setting a breakpoint worked
- ☹ hitting a breakpoint didn't work
  - 🗑 non-runnable/sparse ELF files in YAML form didn't cut it
- 🕮 dealing with tests
  - `yaml2obj`[6] always produced `.symtab`
    - 💣 made my tests go nuts
- ⚖ polishing for upstream

[6] *"yaml2obj takes a YAML description of an object file and converts it to a binary file."* (https://llvm.org/docs/yaml2obj.html)

Real example of sparse ELF file

Check to find symbol `multiplyByFour` in mini-debuginfo

```
# REQUIRES: lzma
# RUN: yaml2obj %s > %t.obj
# RUN: llvm-objcopy --remove-section=.symtab %t.obj
# RUN: %lldb -b -o 'image dump symtab' %t.obj | FileCheck %s
# CHECK: [ 0] 1 X Code 0x0000000004005b0 0x000000000000000f 0x00000012 multiplyByFour

 --- !ELF
FileHeader:
  Class:           ELFCLASS64
  Data:            ELFDATA2LSB
  Type:            ET_EXEC
  Machine:         EM_X86_64
  Entry:           0x00000000004004C0
Sections:
  - Name:            .gnu_debugdata
    Type:            SHT_PROGBITS
    AddressAlign:    0x0000000000000001
    Content:         FD377A585A000004E6 # ...
...
```

- notice line 4 manually removes `.symtab`
- meanwhile `yaml2obj` was fixed

Thank you!

Please, share your feedback ★★★★★

https://submission.fosdem.org/feedback/10393