How to Write a Heap Profiler

Dirty Details Beyond Standard C++

CppCon 2019

presented by Milian Wolff

- Introduction

- Preloading

- Stack Unwinding

- Symbol Resolution

- Runtime Attaching

This talk

- Isn't really about C++

- Demystifies some central tooling aspects

- Contains information I had to learn the hard way

- Is highly platform specific
  - ELF, DWARF, ld-linux, ...

- Shows lots of dirty details
  - Language lawyers beware!

https://github.com/milianw/how-to-write-a-memory-profiler

- heaptrack: heap memory profiler for linux
  - May 2013
  - https://github.com/KDE/heaptrack

- hotspot & perfparser: Linux perf GUI
  - December 2016
  - https://github.com/KDAB/hotspot
  - https://code.qt.io/qt-creator/perfparser.git

- Tracing is a useful debugging and profiling technique
  - perf trace, strace, ltrace, heaptrack, printf-debugging, …

- Requirements for a useful tracer:
  - Cope with thousands or even millions of trace events per second
  - Ideally zero overhead when not used
  - Support for runtime attaching or similar

Need to build a domain specific custom tracer?

- Prefer to use existing tracing frameworks:
  - perf with sdt / uprobe
  - lttng-ust
  - LLVM XRay

So What Do We Want To Achieve?

- Inject custom code into a (running) application

- Intercept all calls to heap (de)allocation functions

- Annotate every (de)allocation event with a backtrace

- Post-process the data to symbolize the backtrace

What is allocating dynamic storage (i.e. heap memory)?

- In `libc`:
  - `malloc`, `free`
  - `realloc`, `calloc`
  - `posix_memalign`, `aligned_alloc`, `valloc`

- In `libstdc++` / `libc++`:
  - `operator new`, `operator new[]`
  - `operator delete`, `operator delete[]`
  - Each with a couple of overloads, e.g. `std::align_val_t`

- In custom allocator implementations:
  - `sbrk`, `mmap` with `MAP_ANONYMOUS`

- Introduction

- **Preloading**

- Stack Unwinding

- Symbol Resolution

- Runtime Attaching

Use the dynamic linker to inject custom library code:

```
LD_PRELOAD=$(readlink -f path/to/libfoo.so) some_app
```

- Dynamic linker resolves library calls

- First library with a suitable exported symbol wins
  - Make sure the mangled name matches

- LD_PRELOAD wins over dynamically linked `libc`

- ODR?
  - C++ standard does not define how linking works

Build a library with the symbols you want to intercept

- Use dlsym from libdl.so to find the original function

- Casting of void* to a function pointer is valid on POSIX

preload.cpp:

```cpp
1  #include <dlfcn.h> // dlsym
2
3  extern "C"
4  {
5  void* malloc(size_t size)
6  {
7      static void* original_malloc = dlsym(RTLD_NEXT, "malloc");
8      assert(original_malloc);
9      auto *original_malloc_fn = reinterpret_cast<decltype(&::malloc)>(original_malloc);
10
11     void *ret = original_malloc_fn(size);
12     fprintf(stderr, "malloc intercepted: %zu -> %p\n", size, ret);
13     return ret;
14 }
15 }
```

```
1  $ nm test_clients/one_malloc | grep " malloc"
2  U malloc
3  $ nm /usr/lib/libc.so.6 | grep " malloc"
4  00000000000875d0 T malloc
5  $ nm preload/libpreload.so | grep " malloc"
6  00000000000875d0 T malloc
```

Preloading

Now we can leverage LD_PRELOAD to inject our custom code:

preload.sh:

```
1  #!/bin/sh
2
3  SCRIPT_PATH=$(readlink -f "$0")
4  SCRIPT_DIR=$(dirname "$SCRIPT_PATH")
5  LIBPRELOAD_PATH=$(readlink -f "$SCRIPT_DIR/libpreload.so")
6
7  # important: we must specify the fully resolved, absolute path
8  # LD_PRELOAD will reject symlinks and relative paths
9  LD_PRELOAD="$LIBPRELOAD_PATH" $@
```

- Use the above script to inject the code into arbitrary applications:

one_malloc.cpp:

```
1  #include <cstdlib>
2
3  int main()
4  {
5      auto *buffer = malloc(100);
6      free(buffer);
7      return buffer ? 0 : 1;
8  }
```

```
1  $ ./preload/preload.sh ./test_clients/one_malloc
2  malloc intercepted: 72704 -> 0x564d8f706260
3  malloc intercepted: 100 -> 0x564d8f717e70
```

Preloading

```
$ LD_DEBUG=bindings ./test_clients/one_malloc |& grep -P '\bmalloc\b'

 1  binding file /usr/lib/libc.so.6 [0] to /usr/lib/libc.so.6 [0]: \
 2          normal symbol `malloc` [GLIBC_2.2.5]
 3  binding file /usr/lib/libgcc_s.so.1 [0] to /usr/lib/libc.so.6 [0]: \
 4          normal symbol `malloc` [GLIBC_2.2.5]
 5  binding file /usr/lib/libstdc++.so.6 [0] to /usr/lib/libc.so.6 [0]: \
 6          normal symbol `malloc` [GLIBC_2.2.5]
 7  binding file /lib64/ld-linux-x86-64.so.2 [0] to /usr/lib/libc.so.6 [0]: \
 8          normal symbol `malloc` [GLIBC_2.2.5]
 9  binding file ./test_clients/one_malloc [0] to /usr/lib/libc.so.6 [0]: \
10          normal symbol `malloc` [GLIBC_2.2.5]
```

Preloading

```
$ LD_DEBUG=bindings ./preload/preload.sh ./test_clients/one_malloc |& grep -P "\bmalloc\b"

 1  binding file /usr/lib/libc.so.6 [0] to .../preload/libpreload.so [0]: \
 2      normal symbol `malloc` [GLIBC_2.2.5]
 3  binding file /usr/lib/libgcc_s.so.1 [0] to .../preload/libpreload.so [0]: \
 4      normal symbol `malloc` [GLIBC_2.2.5]
 5  binding file /usr/lib/libstdc++.so.6 [0] to .../preload/libpreload.so [0]: \
 6      normal symbol `malloc` [GLIBC_2.2.5]
 7  binding file /lib64/ld-linux-x86-64.so.2 [0] to .../preload/libpreload.so [0]: \
 8      normal symbol `malloc` [GLIBC_2.2.5]
 9  binding file .../preload/libpreload.so [0] to /usr/lib/libc.so.6 [0]: \
10      normal symbol `malloc`
11
12  malloc intercepted: 72704 -> 0x5607802f8260
13
14  binding file ./test_clients/one_malloc [0] to .../preload/libpreload.so [0]: \
15      normal symbol `malloc` [GLIBC_2.2.5]
16
17  malloc intercepted: 100 -> 0x560780309e70
```

- Introduction

- Preloading

- **Stack Unwinding**

- Symbol Resolution

- Runtime Attaching

- We need to unwind the stack to get a backtrace

- Approaches:
  - Frame pointer (requires `-fno-omit-frame-pointer`)
  - Exception unwind tables (`.eh_frame` or `.ARM.exidx` sections)
  - DWARF debug information (`.debug_frame` section)
  - Alternatives:
    - Intel LBR (often too shallow)
    - Shadow Stack (https://github.com/nokia/not-perf)

- libc's `backtrace` depends on frame pointers

- elfutils `dwfl_thread_getframes` is complex to use

- libunwind is easy to use, fast and feature rich
  - https://github.com/libunwind/libunwind

Using libunwind is trivial:

```
 1  #define UNW_LOCAL_ONLY
 2  #include <libunwind.h>
 3
 4  std::vector<void*> backtrace()
 5  {
 6      const auto MAX_SIZE = 64;
 7      std::vector<void *> trace(MAX_SIZE);
 8      const auto size = unw_backtrace(trace.data(), MAX_SIZE);
 9      trace.resize(size);
10      return trace;
11  }
```

Using libunwind is trivial:

```
 1  #define UNW_LOCAL_ONLY
 2  #include <libunwind.h>
 3
 4  std::vector<void*> backtrace()
 5  {
 6      const auto MAX_SIZE = 64;
 7      std::vector<void *> trace(MAX_SIZE);
 8      const auto size = unw_backtrace(trace.data(), MAX_SIZE);
 9      trace.resize(size);
10      return trace;
11  }
```

But the output isn't really useable as-is:

```
$ ./backtrace/preload_backtrace.sh ./test_clients/vector
```

```
 1  0x7f37c70ea63c
 2  0x7f37c6f53ac9
 3  0x562c61aa1ca1
 4  0x562c61aa1ad0
 5  0x7f37c6bb4ee2
 6  0x562c61aa1b5d
```

- Introduction

- Preloading

- Stack Unwinding

- **Symbol Resolution**
  - Introduction
  - ELF mappings
  - elfutils
  - Demangling
  - Inline Frames
  - Clang Support

- Runtime Attaching

- Introduction

- ELF mappings

- elfutils

- Demangling

- Inline Frames

- Clang Support

- Goal:
  - From instruction pointer: `0x55b20bc95d9f`
  - To human readable symbol

- Goal:
  - From instruction pointer: `0x55b20bc95d9f`
  - To human readable symbol

```
std::map<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>,
    std::vector<std::map<int, std::__cxx11::basic_string<char, std::char_traits<char>,
    std::allocator<char>>, std::less<int>, std::allocator<std::pair<int const,
    std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>>>>,
    std::allocator<std::map<int, std::__cxx11::basic_string<char, std::char_traits<char>,
    std::allocator<char>>, std::less<int>, std::allocator<std::pair<int const,
    std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>>>>>,
    std::less<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>>,
    std::allocator<std::pair<std::__cxx11::basic_string<char, std::char_traits<char>,
    std::allocator<char>> const, std::vector<std::map<int, std::__cxx11::basic_string<char,
    std::char_traits<char>, std::allocator<char>>, std::less<int>, std::allocator<std::pair<int
    std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>>>>,
    std::allocator<std::map<int, std::__cxx11::basic_string<char, std::char_traits<char>,
    std::allocator<char>>, std::less<int>, std::allocator<std::pair<int const,
    std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>>>>>>
```

- Goal:
  - From instruction pointer: `0x55b20bc95d9f`
  - To human readable symbol

```
std::map<std::string, std::vector<std::map<int, std::string>>>
```

- Goal:
  - From instruction pointer: `0x55b20bc95d9f`
  - To human readable symbol

  ```
  std::map<std::string, std::vector<std::map<int, std::string>>>
  ```

- The last step requires heuristics
  - Typedefs will change based on used STL implementation

- Map instruction pointer address to ELF offset:

```
1  $ ./backtrace/preload_backtrace.sh ./test_clients/delay &
2
3  $ cat /proc/$(pidof delay)/maps
```

- Find first address for file that has mapping which includes `0x55b20bc95d9f`:

```
1  ...
2  55b20bc95000-55b20bc97000 r-xp 00000000 08:04 18622503 \
3       .../test_clients/delay
4  55b20bc97000-55b20bc98000 r--p 00001000 08:04 18622503 \
5       .../test_clients/delay
6  55b20bc98000-55b20bc99000 rw-p 00002000 08:04 18622503 \
7       .../test_clients/delay
8  ...
```

- Mapped address: `0x55b20bc95d9f - 0x55b20bc95000 = 0xd9f`

- Now finally resolve the symbol:

```
$ addr2line -p -e .../test_clients/delay -a 0xd9F
```

```
1  0x0000000000000d9f:
2      /usr/include/c++/9.1.0/ostream:570
```

- Now finally resolve the symbol:

```
$ addr2line -p -e .../test_clients/delay -a 0xd9F
```

```
1  0x0000000000000d9f:
2      /usr/include/c++/9.1.0/ostream:570
```

```
$ addr2line -p -f -e .../test_clients/delay -a 0xd9f # with function names
```

```
1  0x0000000000000d9f:
2      _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
3          at /usr/include/c++/9.1.0/ostream:570
```

Introduction

- Now finally resolve the symbol:

```
$ addr2line -p -e .../test_clients/delay -a 0xd9F
```

```
1  0x0000000000000d9f:
2      /usr/include/c++/9.1.0/ostream:570
```

```
$ addr2line -p -f -e .../test_clients/delay -a 0xd9f # with function names
```

```
1  0x0000000000000d9f:
2      _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
3          at /usr/include/c++/9.1.0/ostream:570
```

```
$ addr2line -p -f -C -e .../test_clients/delay -a 0xd9f # with demangling
```

```
1  0x0000000000000d9f:
2      std::basic_ostream<...>& std::operator<< <...>(...)
3          at /usr/include/c++/9.1.0/ostream:570
```

- Now finally resolve the symbol:

```
$ addr2line -p -e .../test_clients/delay -a 0xd9F
```

```
1  0x0000000000000d9f:
2       /usr/include/c++/9.1.0/ostream:570
```

```
$ addr2line -p -f -e .../test_clients/delay -a 0xd9f # with function names
```

```
1  0x0000000000000d9f:
2       _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
3           at /usr/include/c++/9.1.0/ostream:570
```

```
$ addr2line -p -f -C -e .../test_clients/delay -a 0xd9f # with demangling
```

```
1  0x0000000000000d9f:
2       std::basic_ostream<...>& std::operator<< <...>(...)
3           at /usr/include/c++/9.1.0/ostream:570
```

```
$ addr2line -p -f -C -i -e .../test_clients/delay -a 0xd9f # with inlines
```

```
1  0x0000000000000d9f:
2       std::basic_ostream<...>& std::operator<< <...>(...)
3           at /usr/include/c++/9.1.0/ostream:570
4       (inlined by) main
5           at .../test_clients/delay.cpp:9
```

Introduction

- Introduction

- **ELF mappings**

- elfutils

- Demangling

- Inline Frames

- Clang Support

Iterate over DSO mappings with dl_iterate_phdr from libdl.so / link.h:

```cpp
 1  #include <link.h>
 2
 3  void dumpMappings(FILE *out)
 4  {
 5      dl_iterate_phdr([](dl_phdr_info *info, size_t /*size*/, void *data) -> int {
 6          auto *name = info->dlpi_name;
 7          if (!name || !name[0])
 8              name = "exe";
 9
10          auto out = reinterpret_cast<FILE *>(data);
11          fprintf(out, "%s is mapped at: 0x%zx\n", name, info->dlpi_addr);
12          return 0;
13      }, out);
14  }
```

Integrated into LD_PRELOAD library:

- Also intercept `dlopen` and `dlclose`
  - Mark mappings as dirty whenever one of these is called

- Before writing a backtrace, check if mapping cache is dirty
  - If so, iterate the current mappings and output it

- On startup, also output the executable path once
  - `std::filesystem::canonical("/proc/self/exe")`

  `./mappings/preload_mappings.sh` `./test_clients/one_malloc`

ELF mappings

```
 1  exe: .../test_clients/vector
 2  begin modules
 3      module: 0x5569a6bce000 exe
 4      module: 0x7fffc0fff000 linux-vdso.so.1
 5      module: 0x7f81ed72d000 .../mappings/libpreload_mappings.so
 6      module: 0x7f81ed4f5000 /usr/lib/libstdc++.so.6
 7      module: 0x7f81ed3af000 /usr/lib/libm.so.6
 8      module: 0x7f81ed395000 /usr/lib/libgcc_s.so.1
 9      module: 0x7f81ed1d2000 /usr/lib/libc.so.6
10      module: 0x7f81ed1cd000 /usr/lib/libdl.so.2
11      module: 0x7f81ed1a9000 /usr/lib/libunwind.so.8
12      module: 0x7f81ed735000 /lib64/ld-linux-x86-64.so.2
13      module: 0x7f81ecf83000 /usr/lib/liblzma.so.5
14      module: 0x7f81ecf62000 /usr/lib/libpthread.so.0
15  end modules
16  malloc(72704) = 0x5569a8081620
17      ip: 0x7f81ed72f26c
18      ip: 0x7f81ed593aea
19      ip: 0x7f81ed745799
20      ip: 0x7f81ed7458a0
21      ip: 0x7f81ed737139
22  malloc(4) = 0x5569a8080400
23  ...
```

ELF mappings

- Introduction

- ELF mappings

- **elfutils**

- Demangling

- Inline Frames

- Clang Support

- Doing symbol resolution is complex
  - Symbol table
  - Debug information interpretation
  - Compressed debug information
  - Split debug info

- Let's use `libdwfl` from elfutils for this task

Basic libdwfl setup:

symbolizer.cpp

```
 1  #include <libdwfl.h>
 2
 3  namespace {
 4  const Dwfl_Callbacks s_callbacks = {
 5      /* find_elf = */        &dwfl_build_id_find_elf,
 6      /* find_debuginfo = */  &dwfl_standard_find_debuginfo,
 7      /* section_address = */ &dwfl_offline_section_address,
 8      /* debuginfo_path = */  nullptr
 9  };
10  }
11
12  Symbolizer::Symbolizer()
13      : m_dwfl(dwfl_begin(&s_callbacks))
14  {
15  }
16
17  Symbolizer::~Symbolizer()
18  {
19      dwfl_end(m_dwfl);
20  }
```

elfutils

Reporting mapped ELF objects:

symbolizer.cpp

```cpp
 1  void Symbolizer::beginReportElf()
 2  {
 3      dwfl_report_begin(m_dwfl);
 4  }
 5
 6  // addr is the base address obtained from e.g. dl_phdr_info->dlpi_addr
 7  void Symbolizer::reportElf(const std::string &path, uint64_t addr)
 8  {
 9      if (!dwfl_report_elf(m_dwfl, path.c_str(), path.c_str(), -1, addr, false)) {
10          fprintf(stderr, "failed to report elf %s at %zx: %s\n",
11                  path.c_str(), addr, dwfl_errmsg(dwfl_errno()));
12      }
13  }
14
15  void Symbolizer::endReportElf()
16  {
17      if (dwfl_report_end(m_dwfl, nullptr, nullptr) != 0) {
18          fprintf(stderr, "failed to end elf reporting: %s\n",
19                  dwfl_errmsg(dwfl_errno()));
20      }
21  }
```

Resolve symbol of address:

symbolizer.cpp

```cpp
 1  Symbol Symbolizer::symbol(uint64_t ip)
 2  {
 3      auto *mod = dwfl_addrmodule(m_dwfl, ip);
 4      if (!mod) {
 5          fprintf(stderr, "failed to find module for ip %zx: %s\n",
 6                  ip, dwfl_errmsg(dwfl_errno()));
 7          return {};
 8      }
 9
10      Symbol symbol;
11      setDsoInfo(symbol, mod, ip);
12      setSymInfo(symbol, mod, ip);
13      setFileLineInfo(symbol, mod, ip);
14      return symbol;
15  }
```

```cpp
 1  // 0xDEADBEEF -> libfoo @ 0xBEEF
 2  void setDsoInfo(Symbol &symbol, Dwfl_Module *mod, Dwarf_Addr ip)
 3  {
 4      Dwarf_Addr moduleStart = 0;
 5      symbol.dso = dwfl_module_info(mod, nullptr, &moduleStart, nullptr,
 6                                    nullptr, nullptr, nullptr, nullptr);
 7      symbol.dso_offset = ip - moduleStart;
 8  }
```

elfutils

Resolving symbol names from symbol table or DWARF:

symbolizer.cpp

```
 1  // 0xDEADBEEF -> foobar @ 0xEF
 2  void setSymInfo(Symbol &symbol, Dwfl_Module *mod, Dwarf_Addr ip)
 3  {
 4      GElf_Sym sym;
 5      auto symname = dwfl_module_addrinfo(mod, ip, &symbol.offset, &sym,
 6                                          nullptr, nullptr, nullptr);
 7      if (!symname)
 8          symname = "??";
 9      symbol.name = symname;
10  }
```

libdwfl: Resolving Source Code Locations

Resolving source code file name and line number from DWARF:

symbolizer.cpp

```cpp
 1  // 0xDEADBEEF -> foo.cpp:42
 2  void setFileLineInfo(Symbol &symbol, Dwfl_Module *mod, Dwarf_Addr ip)
 3  {
 4      Dwarf_Addr bias = 0;
 5      auto die = dwfl_module_addrdie(mod, ip, &bias);
 6      if (!die)
 7          return;
 8      auto srcloc = dwarf_getsrc_die(die, ip - bias);
 9      if (!srcloc)
10          return;
11      auto srcfile = dwarf_linesrc(srcloc, nullptr, nullptr);
12      if (!srcfile)
13          return;
14
15      symbol.file = srcfile;
16      dwarf_lineno(srcloc, &symbol.line);
17      dwarf_linecol(srcloc, &symbol.column);
18  }
```

elfutils

- Introduction

- ELF mappings

- elfutils

- **Demangling**

- Inline Frames

- Clang Support

Demangling C++ function with c++filt:

```
$ c++filt _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc

1  std::basic_ostream<char, std::char_traits<char>>&
2      std::operator<< <std::char_traits<char>>
3      (std::basic_ostream<char, std::char_traits<char>>&, char const*)
```

Manual demangling via `cxxabi.h`:

symbolizer.cpp

```cpp
 1  #include <cxxabi.h>
 2
 3  std::string Symbolizer::demangle(const std::string &symbol) const
 4  {
 5      if (symbol.size() < 3 || symbol[0] != '_' || symbol[1] != 'Z')
 6          return symbol;
 7      auto demangled = abi::__cxa_demangle(symbol.c_str(), nullptr,
 8                                           nullptr, nullptr);
 9      if (!demangled)
10          return symbol;
11      std::string ret = demangled;
12      free(demangled);
13      return ret;
14  }
```

Putting it all together:

```
$ ./mappings/preload_mappings.sh ./test_clients/vector |& ./symbolization/symbolization

 1 ...
 2 malloc(4) = 0x55f390225400
 3 ip: 0x7fbaebf6d26c
 4     intercept::malloc(unsigned long)@2c
 5     .../mappings/libpreload_mappings.so@226c
 6     .../src/shared/alloc_hooks.h:85:18
 7 ip: 0x7fbaebdd5ac9
 8     operator new(unsigned long)@19
 9     /usr/lib/libstdc++.so.6@a2ac9
10     /build/gcc/src/gcc/libstdc++-v3/libsupc++/new_op.cc:50:22
11 ip: 0x55f38ef73ca1
12     void std::vector<int, std::allocator<int> >::_M_realloc_insert...
13     .../test_clients/vector@ca1
14     /usr/include/c++/9.1.0/ext/new_allocator.h:114:41
15 ip:0x55f38ef73ad0
16     main@70
17     .../test_clients/vector@ad0
18     /usr/include/c++/9.1.0/bits/vector.tcc:121:4
19 ip: 0x7fbaeba36ee2
20     __libc_start_main@f2
21     /usr/lib/libc.so.6@26ee2
22 ip: 0x55f38ef73b5d
23     _start@2d
24     .../test_clients/vector@b5d
```

Symbol Resolution

- Introduction

- ELF mappings

- elfutils

- Demangling

- **Inline Frames**

- Clang Support

Find compilation unit (CU) debug information entry (DIE):

symbolization.cpp

```cpp
1  auto *mod = dwfl_addrmodule(m_dwfl, ip);
2  if (!mod)
3      return {};
4
5  Dwarf_Addr bias = 0;
6  // CU DIE: Compilation Unit Debug Information Entry
7  auto cuDie = dwfl_module_addrdie(mod, ip, &bias);
8  if (!cuDie)
9      return {};
```

Inline Frames

Find innermost scope DIE:

symbolization.cpp

```
 1  // innermost scope DIE
 2  Dwarf_Die scopeDie;
 3  {
 4      Dwarf_Die *scopes = nullptr;
 5      const auto nscopes = dwarf_getscopes(cuDie, ip - bias, &scopes);
 6      if (nscopes == 0)
 7          return {};
 8      scopeDie = scopes[0];
 9      free(scopes);
10  }
```

Inline Frames

Find inlined frames, i.e. other DIEs that contain scope DIE:

symbolization.cpp

```cpp
 1  Dwarf_Die *scopes = nullptr;
 2  const auto nscopes = dwarf_getscopes_die(&scopeDie, &scopes);
 3
 4  Dwarf_Files *files = nullptr;
 5  dwarf_getsrcfiles(cuDie, &files, nullptr);
 6
 7  std::vector<Symbol> symbols;
 8  for (int i = 0; i < nscopes; ++i) {
 9      const auto scope = &scopes[i];
10      if (dwarf_tag(scope) == DW_TAG_inlined_subroutine)
11          symbols.push_back(inlinedSubroutineSymbol(scope, files));
12  }
13  free(scopes);
```

libdwfl:Symbol Resolution for Inlined Frames

Resolve symbol name for inlined frame scope:

symbolization.cpp

```
 1  const char *dieName(Dwarf_Die *die)
 2  {
 3      Dwarf_Attribute attr;
 4      Dwarf_Attribute *result = dwarf_attr_integrate(die, DW_AT_MIPS_linkage_name, &attr);
 5      if (!result)
 6          result = dwarf_attr_integrate(die, DW_AT_linkage_name, &attr);
 7
 8      auto *name = dwarf_formstring(result);
 9      if (name)
10          return name;
11
12      return dwarf_diename(die);
13  }
```

Inline Frames

Resolve source file and line number for inlined frame scope:

symbolization.cpp

```cpp
 1  Symbol inlinedSubroutineSymbol(Dwarf_Die *scope, Dwarf_Files *files)
 2  {
 3      Dwarf_Attribute attr;
 4      Dwarf_Word val = 0;
 5
 6      Symbol symbol;
 7
 8      symbol.name = dieName(scope);
 9
10      const char *file = nullptr;
11      if (dwarf_formudata(dwarf_attr(scope, DW_AT_call_file, &attr), &val) == 0)
12          file = dwarf_filesrc(files, val, nullptr, nullptr);
13      symbol.file = file ? file : "??";
14
15      if (dwarf_formudata(dwarf_attr(scope, DW_AT_call_line, &attr), &val) == 0)
16          symbol.line = static_cast<int>(val);
17
18      if (dwarf_formudata(dwarf_attr(scope, DW_AT_call_column, &attr), &val) == 0)
19          symbol.column = static_cast<int>(val);
20
21      return symbol;
22  }
```

Inline Frames

Symbolizing Allocation Backtraces

Putting it all together:

```
$ ./mappings/preload_mappings.sh ./test_clients/vector |& \
    ./symbolization_inlines/symbolization_inlines

 1  malloc(4) = 0x55de17384400
 2  ip: 0x7fb788cd026c (.../mappings/libpreload_mappings.so@226c)
 3      intercept::malloc(unsigned long)@2c
 4  ip: 0x7fb788b38ac9 (/usr/lib/libstdc++.so.6@a2ac9)
 5      operator new(unsigned long)@19
 6  ip: 0x55de15b09ca1 (.../test_clients/vector@ca1)
 7      inline: __gnu_cxx::new_allocator<int>::allocate(unsigned long, void const*)
 8      inline: std::allocator_traits<std::allocator<int> >::allocate...
 9      inline: std::_Vector_base<int, std::allocator<int> >::_M_allocate(unsigned long)
10      void std::vector<int, std::allocator<int> >::_M_realloc_insert<int>(...
11  ip: 0x55de15b09ad0 (.../test_clients/vector@ad0)
12      inline: int& std::vector<int, std::allocator<int> >::emplace_back<int>(int&&)
13      inline: std::vector<int, std::allocator<int> >::push_back(int&&)
14      inline: std::back_insert_iterator<std::vector<int, std::allocator<int> > >::operator=...
15      inline: generate_n<std::back_insert_iterator<std::vector<int> >, int, ...
16      main@70
17  ip: 0x7fb788799ee2 (/usr/lib/libc.so.6@26ee2)
18      __libc_start_main@f2
19  ip: 0x55de15b09b5d (.../test_clients/vector@b5d)
20      _start@2d
```

Inline Frames

- Introduction

- ELF mappings

- elfutils

- Demangling

- Inline Frames

- **Clang Support**

- Clang does not emit `.debug_aranges` section by default
  - This breaks CU DIE lookup via `dwfl_module_addrdie`
  - Symbolization fails to find source file information, inline frames

- Option 1:
  - Recompile everything with `clang++ -gdwarf-aranges ...`

- Option 2: workaround by building the mapping manually
  - Based on https://github.com/bombela/backward-cpp
  - `dwfl_module_nextcu` to find all CU DIEs in a module
  - `dwarf_child` to find all DIEs in the CU DIE
  - `dwarf_ranges` to find ranges for DIE
  - See `moduleAddrDie` in `symbolization_clang/symbolizer.cpp`

- Introduction

- Preloading

- Stack Unwinding

- Symbol Resolution

- **Runtime Attaching**
  - Code Injection
  - Intercepting Library Calls

- Code Injection

- Intercepting Library Calls

Q: How can we inject and execute our code into a running application?

Q: How can we inject and execute our code into a running application?

A: With a debugger and dlopen!

Caveat: only works for dynamically linked applications

- Attach to any application via `gdb -p PID`
  - Auto loading of symbols is often quite slow
  - Thus disable that and resolve necessary symbols manually

- Call `dlopen` to load your own code
  - `dlopen` is only available when application links against `libdl.so`
  - Use `libc.so` internal `__libc_dlopen_mode` instead

- Optionally call custom init function with tracing arguments

Code Injection

- Attach to any application via gdb -p PID
  - Auto loading of symbols is often quite slow
  - Thus disable that and resolve necessary symbols manually

- Call dlopen to load your own code
  - dlopen is only available when application links against libdl.so
  - Use libc.so internal __libc_dlopen_mode instead

- Optionally call custom init function with tracing arguments

```
 1  __RTLD_DLOPEN="0x80000000"
 2  RTLD_NOW="0x00002"
 3  gdb --batch-silent -n \
 4      -iex="set auto-solib-add off" \
 5      -p $pid \
 6      --eval-command="sharedlibrary libc.so" \
 7      --eval-command="call (void) __libc_dlopen_mode(\"/path/to/mylib.so\", \
 8                                                      $__RTLD_DLOPEN | $RTLD_NOW)" \
 9      --eval-command="sharedlibrary mylib" \
10      --eval-command="call (void) attach_init(\"/path/to/trace.log\")" \
11      --eval-command="detach"
```

Code Injection

- Code Injection

- Intercepting Library Calls

Short Introduction to GOT / PLT

- Calls to dynamically shared objects require relocations
  - See: https://www.akkadia.org/drepper/dsohowto.pdf

- GOT: Global Offset Table
  - Writable section for the linker

- PLT: Procedure Linkage Table

```
1  int main() {
2      auto *buffer = malloc(100);
3  }
```

Relocations can be seen with readelf:

```
$ readelf -r test_clients/one_malloc
```

```
1  Relocation section '.rela.plt' at offset 0x6b8 contains 2 entries:
2  Offset          Info           Type           Sym. Value    Sym. Name + Addend
3  000000002000  000100000007 R_X86_64_JUMP_SLO 0000000000000000 malloc@GLIBC_2.2.5 + 0
```

Indirect calls to malloc are visible in the disassembly:

```
$ objdump -S test_clients/one_malloc
```

```
1  ...
2  0000000000000720 <malloc@plt>:                                  # disassembly of section .plt
3  720:   ff 25 da 18 00 00       jmpq   *0x18da(%rip)        # 2000 <malloc@GLIBC_2.2.5>
4  726:   68 00 00 00 00          pushq  $0x0
5  72b:   e9 e0 ff ff ff          jmpq   710 <_init+0x28>
6  ...
7  0000000000000839 <main>:                                        # disassembly of section .text
8  ...
9      auto *buffer = malloc(100);
10 841:   bf 64 00 00 00          mov    $0x64,%edi
11 846:   e8 d5 fe ff ff          callq  720 <malloc@plt>
12 84b:   48 89 45 f8             mov    %rax,-0x8(%rbp)
```

Intercepting Library Calls

- Iterate over all relocations in all DSOs

- Check if the symbol name matches one of our trace points

- If so, overwrite the address in the GOT with the address to our trace point

Basic example:

got_overwriting.cpp

```
 1  void *foo()
 2  {
 3      return malloc(100);
 4  }
 5
 6  int main()
 7  {
 8      auto f = foo();
 9      overwritePhdrs();
10      auto f2 = foo();
11
12      free(f);
13      free(f2);
14      return f != f2;
15  }
```

Dynamic Rebinding: Dynamic Sections

Iterate over dynamic sections:

got_overwriting.cpp

```cpp
 1  void overwritePhdrs()
 2  {
 3      dl_iterate_phdr([](dl_phdr_info* info, size_t /*size*/, void* /*data*/) {
 4          if (strstr(info->dlpi_name, "/ld-linux")) {
 5              // don't touch anything in the linker itself
 6              return 0;
 7          }
 8
 9          for (int i = 0; i < info->dlpi_phnum; ++i) {
10              const auto &phdr = info->dlpi_phdr[i];
11              if (phdr.p_type == PT_DYNAMIC) {
12                  const auto dynEntriesAddr = phdr.p_vaddr + info->dlpi_addr;
13                  const auto dynEntries = reinterpret_cast<const Elf::Dyn *>(dynEntriesAddr);
14                  const auto base = info->dlpi_addr;
15                  overwriteDynEntries(dynEntries, base);
16              }
17          }
18          return 0;
19      }, nullptr);
20  }
```

Initializing access to ELF tables:

got_overwriting.cpp

```cpp
 1  void overwriteDynEntries(const Elf::Dyn *dynEntries, Elf::Addr baseAddr)
 2  {
 3      Elf::SymbolTable symbols;
 4      Elf::StringTable strings;
 5      Elf::JmprelTable jmprels;
 6
 7      // initialize the elf tables
 8      for (auto dyn = dynEntries; dyn->d_tag != DT_NULL; ++dyn) {
 9          symbols.consume(dyn) || strings.consume(dyn) || jmprels.consume(dyn) ;
10      }
11
12      overwriteGotEntries(jmprels, symbols, strings, baseAddr);
13      // TODO: also do this for REL and RELA tables
14  }
```

Required ELF types:

elftable.h

```cpp
 1  using Addr = ElfW(Addr);
 2  using Dyn = ElfW(Dyn);
 3  using Rel = ElfW(Rel);
 4  using Rela = ElfW(Rela);
 5  using Sym = ElfW(Sym);
 6  using Sxword = ElfW(Sxword);
 7  using Xword = ElfW(Xword);
 8
 9  using StringTable = Table<const char, DT_STRTAB, DT_STRSZ>;
10  using SymbolTable = Table<Elf::Sym, DT_SYMTAB, DT_SYMENT>;
11  using RelTable = Table<Elf::Rel, DT_REL, DT_RELSZ>;
12  using RelaTable = Table<Elf::Rela, DT_RELA, DT_RELASZ>;
13  using JmprelTable = Table<Elf::Rela, DT_JMPREL, DT_PLTRELSZ>;
```

Accessing ELF tables:

elftable.h

```cpp
 1  template <typename T, Elf::Sxword AddrTag, Elf::Sxword SizeTag>
 2  struct Table
 3  {
 4      using type = T;
 5      T* table = nullptr;
 6      Elf::Xword size = {};
 7
 8      bool consume(const Elf::Dyn* dyn) noexcept
 9      {
10          if (dyn->d_tag == AddrTag) {
11              table = reinterpret_cast<T*>(dyn->d_un.d_ptr);
12              return true;
13          } else if (dyn->d_tag == SizeTag) {
14              size = dyn->d_un.d_val;
15              return true;
16          }
17          return false;
18      }
19
20      const T* begin() const noexcept { return table; }
21      const T* end() const noexcept { return table + size / sizeof(T); }
22  };
```

Iterate over relocations:

got_overwriting.cpp

```cpp
 1  void overwriteGotEntries(const Elf::JmprelTable &relocations,
 2                           const Elf::SymbolTable &symbols,
 3                           const Elf::StringTable &strings,
 4                           Elf::Addr baseAddr)
 5  {
 6      for (const auto &relocation : relocations) {
 7          const auto index = ELF_R_SYM(relocation.r_info);
 8          const char* symname = strings.table + symbols.table[index].st_name;
 9          const auto gotAddr = relocation.r_offset + baseAddr;
10          overwriteGotEntry(symname, gotAddr);
11      }
12  }
```

Overwrite the GOT entry to intercept function calls:

got_overwriting.cpp

```cpp
 1  void *intercept_malloc(size_t size)
 2  {
 3      static auto original_malloc = dlsym(RTLD_NEXT, "malloc");
 4      assert(original_malloc);
 5
 6      auto original_malloc_fn = reinterpret_cast<decltype(&::malloc)>(original_malloc);
 7      auto ret = original_malloc_fn(size);
 8
 9      fprintf(stderr, "malloc intercepted: %zu -> %p\n", size, ret);
10      return ret;
11  }
12
13  void overwriteGotEntry(const char *symname, Elf::Addr gotAddr)
14  {
15      if (strcmp(symname, "malloc") == 0) {
16          auto ptr = reinterpret_cast<void **>(gotAddr);
17          fprintf(stderr, "relocation: %s: %zx | %p\n", symname, gotAddr, *ptr);
18          *ptr = reinterpret_cast<void *>(&intercept_malloc);
19      }
20  }
```

Putting it all together:

got_overwriting

```
1 $ ./got_overwriting/got_overwriting
2 relocation: malloc: 5604cc5c0038 | 0x7fec64a0e5d0
3 malloc intercepted: 100 -> 0x5604cd0a8ee0
```

Putting it all together:

attach_intercept

```
1  $ ./attach_intercept/attach_intercept.sh $(pidof ...) \
2      |& ./symbolization_inlines/symbolization_inlines
```

```
 1  ...
 2  malloc(8) = 0x563360be0f40
 3  ip: 0x7f906790f67c (.../attach_intercept/libattach_intercept.so@267c)
 4      intercept::malloc(unsigned long)@2c
 5  ip: 0x7f906777dac9 (/usr/lib/libstdc++.so.6@a2ac9)
 6      operator new(unsigned long)@19
 7  ip: 0x56335f93b041
 8      __gnu_cxx::new_allocator<double>::allocate(unsigned long, void const*)
 9      std::allocator_traits<std::allocator<double> >::allocate(std::allocator<double>&, ...
10      std::_Vector_base<double, std::allocator<double> >::_M_allocate(unsigned long)
11      void std::vector<double, std::allocator<double> >::_M_realloc_insert<double>(...
12  ip: 0x56335f93ae3a (.../test_clients/delay@e3a)
13      double& std::vector<double, std::allocator<double> >::emplace_back<double>(double&&)
14      std::vector<double, std::allocator<double> >::push_back(double&&)
15      std::back_insert_iterator<std::vector<double, std::allocator<double> > >::operator=(...
16      generate_n<std::back_insert_iterator<std::vector<double> >, long unsigned int, ...
17      main@ca
18  ip: 0x7f90673deee2 (/usr/lib/libc.so.6@26ee2)
19      __libc_start_main@f2
20  ip: 0x56335f93aefd (.../test_clients/delay@efd)
21      _start@2d
22  ...
```

Intercepting Library Calls

- Efficient output data format
  - Consider zstd compression
  - Consider binary data format
  - Graphing essentially requires a good time series data format

- Handling other quirks during record time
  - forking, sub processes, clean shutdown, ...

- API for custom allocators

- Analysis GUI
  - FlameGraph visualization is crucial

Please contribute to heaptrack instead of writing your own!

Questions?

Milian Wolff

- milian.wolff@kdab.com
- Example source code and slides
- Heap Memory Profiler: heaptrack
- Linux Perf GUI: hotspot