How to Write a Heap Profiler

Dirty Details Beyond Standard C++

CppCon 2019

presented by Milian Wolff

- How to Write a Heap Profiler
  - Introduction
  - Preloading
  - Stack Unwinding
  - Symbol Resolution
  - Runtime Attaching

- Introduction
- Preloading
- Stack Unwinding
- Symbol Resolution
- Runtime Attaching

This talk

- Is highly platform specific
  - ELF, DWARF, ld-linux, ...
- Shows lots of dirty tricks
  - Language lawyers beware!
- Contains information I had to learn the hard way

How I Learned All This

- heaptrack: heap memory profiler for linux
  - https://github.com/KDE/heaptrack
  - In-process backtrace generation with libunwind
  - Runtime attaching

- perfparser & hotspot: Linux perf GUI
  - https://github.com/KDAB/hotspot
  - Out-of-process symbol resolution with elfutils / libdwfl

Tracing

- 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
  - Close to zero overhead when not used
  - Runtime attaching or similar

Need to build a domain specific custom tracer?

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

- Introduction
- **Preloading**
- Stack Unwinding
- Symbol Resolution
- Runtime Attaching

Memory Allocations

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[]`
  - `std::align_val_t` overloads
- In custom allocator implementations:
  - `sbrk`, `mmap` with `MAP_ANONYMOUS`

LD_PRELOAD

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

Intercepting Library Calls

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
#include <dlfcn.h> // dlsym

extern "C"
{
void* malloc(size_t size) noexcept
{
    static auto original_malloc = dlsym(RTLD_NEXT, "malloc");
    assert(original_malloc);
    auto original_malloc_fn = reinterpret_cast<decltype(&::malloc)>(original_malloc);

    auto *ret = original_malloc_fn(size);
    fprintf(stderr, "malloc intercepted: %zu -> %p\n", size, ret);
    return ret;
}
}
```

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

Intercepting Library Calls (cont'd)

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
```

Inspecting Dynamic Linking: Without LD_PRELOAD

```
$ 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]
```

Inspecting Dynamic Linking: With LD_PRELOAD

```
$ 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 unwinding (requires `-fno-omit-frame-pointer`)
  - Use 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)

Backtrace Libraries

- 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**
  - ELF mappings
  - elfutils
  - Demangling
  - Inline Frames
  - Clang Support
- Runtime Attaching

- Instruction pointer: `0x55b20bc95d9f`
- Corresponding ELF map:

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

```
55b20bc95000-55b20bc97000 r-xp 00000000 08:04 18622503 \
    .../test_clients/delay
55b20bc97000-55b20bc98000 r--p 00001000 08:04 18622503 \
    .../test_clients/delay
55b20bc98000-55b20bc99000 rw-p 00002000 08:04 18622503 \
    .../test_clients/delay
...
```

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

```
$ addr2line -p -e .../test_clients/delay -a 0xD9F
```

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

- Instruction pointer: `0x55b20bc95d9f`
- Corresponding ELF map:

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

```
55b20bc95000-55b20bc97000 r-xp 00000000 08:04 18622503 \
    .../test_clients/delay
55b20bc97000-55b20bc98000 r--p 00001000 08:04 18622503 \
    .../test_clients/delay
55b20bc98000-55b20bc99000 rw-p 00002000 08:04 18622503 \
    .../test_clients/delay
...
```

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

```
$ addr2line -p -f -e .../test_clients/delay -a 0xD9F
```

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

- Instruction pointer: `0x55b20bc95d9f`
- Corresponding ELF map:

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

```
55b20bc95000-55b20bc97000 r-xp 00000000 08:04 18622503 \
    .../test_clients/delay
55b20bc97000-55b20bc98000 r--p 00001000 08:04 18622503 \
    .../test_clients/delay
55b20bc98000-55b20bc99000 rw-p 00002000 08:04 18622503 \
    .../test_clients/delay
...
```

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

```
$ addr2line -p -f -C -e .../test_clients/delay -a 0xD9F
```

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

- Instruction pointer: `0x55b20bc95d9f`
- Corresponding ELF map:

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

```
55b20bc95000-55b20bc97000 r-xp 00000000 08:04 18622503 \
    .../test_clients/delay
55b20bc97000-55b20bc98000 r--p 00001000 08:04 18622503 \
    .../test_clients/delay
55b20bc98000-55b20bc99000 rw-p 00002000 08:04 18622503 \
    .../test_clients/delay
...
```

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

```
$ addr2line -p -f -C -i -e .../test_clients/delay -a 0xD9F
```

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

- ELF mappings
- elfutils
- Demangling
- Inline Frames
- Clang Support

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

```
 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 }
```

Preload Mappings

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
```

Preload Mappings Output

```
exe: .../test_clients/vector
begin modules
    module: 0x5569a6bce000 exe
    module: 0x7fffc0fff000 linux-vdso.so.1
    module: 0x7f81ed72d000 .../mappings/libpreload_mappings.so
    module: 0x7f81ed4f5000 /usr/lib/libstdc++.so.6
    module: 0x7f81ed3af000 /usr/lib/libm.so.6
    module: 0x7f81ed395000 /usr/lib/libgcc_s.so.1
    module: 0x7f81ed1d2000 /usr/lib/libc.so.6
    module: 0x7f81ed1cd000 /usr/lib/libdl.so.2
    module: 0x7f81ed1a9000 /usr/lib/libunwind.so.8
    module: 0x7f81ed735000 /lib64/ld-linux-x86-64.so.2
    module: 0x7f81ecf83000 /usr/lib/liblzma.so.5
    module: 0x7f81ecf62000 /usr/lib/libpthread.so.0
end modules
malloc(72704) = 0x5569a8081620
    ip: 0x7f81ed72f26c
    ip: 0x7f81ed593aea
    ip: 0x7f81ed745799
    ip: 0x7f81ed7458a0
    ip: 0x7f81ed737139
malloc(4) = 0x5569a8080400
...
```

Symbol Resolution

- ELF mappings
- elfutils
- Demangling
- Inline Frames
- Clang Support

Symbol Resolution with elfutils

- 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     &dwfl_build_id_find_elf,
 6     &dwfl_standard_find_debuginfo,
 7     &dwfl_offline_section_address,
 8     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 }
```

Reporting mapped ELF objects:

symbolizer.cpp

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

Resolve symbol of address:

symbolizer.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 }
```

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

libdwfl: Resolving Symbol Names

Resolving symbol names from symbol table or DWARF:

symbolizer.cpp

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

libdwfl: Resolving Source Code Locations

Resolving source code file name and line number from DWARF:

symbolizer.cpp

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

- 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*)
```

Demangling: cxxabi.h

Manual demangling via cxxabi.h:

symbolizer.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
```

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

- ELF mappings
- elfutils
- Demangling
- **Inline Frames**
- Clang Support

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

symbolization.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 {};
```

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 }
```

libdwfl: Inlined DIEs

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

symbolization.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 }
```

libdwfl: Information for inlined DIE

Resolve source file and line number for inlined frame scope:

symbolization.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 }
```

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 /usr/include/c++/9.1.0/bits/vector.tcc:121:4
17 ip: 0x7fb788799ee2 (/usr/lib/libc.so.6@26ee2)
18     __libc_start_main@f2
19 ip: 0x55de15b09b5d (.../test_clients/vector@b5d)
20     _start@2d
```

- ELF mappings
- elfutils
- Demangling
- Inline Frames
- Clang Support

Clang: Missing `.debug_aranges`

- 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

GDB 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

GDB 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
- 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

Short Introduction to GOT / PLT (cont'd)

```
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  Disassembly of section .plt:
2  ...
3  0000000000000720 :
4  720:   ff 25 da 18 00 00       jmpq   *0x18da(%rip)        # 2000
5  726:   68 00 00 00 00          pushq  $0x0
6  72b:   e9 e0 ff ff ff          jmpq   710
7  ...
8  0000000000000839 :
9  ...
10     auto *buffer = malloc(100);
11 841:   bf 64 00 00 00          mov    $0x64,%edi
12 846:   e8 d5 fe ff ff          callq  720
13 84b:   48 89 45 f8             mov    %rax,-0x8(%rbp)
```

Dynamic Rebinding For Tracing

- 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

Dynamic Rebinding For Tracing

- 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

```
 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 For Tracing

- 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

```
 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 }
```

Dynamic Rebinding For Tracing

- 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

```
 1 void overwriteDynEntries(const Elf::Dyn *dynEntries, Elf::Addr baseAddr)
 2 {
 3     Elf::SymbolTable symbols;
 4     Elf::JmprelTable jmprels;
 5     Elf::StringTable strings;
 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 }
```

Dynamic Rebinding For Tracing

- 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

```
 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 };
```

Dynamic Rebinding For Tracing

- 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

```
 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>;
```

Dynamic Rebinding For Tracing

- 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

```
 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 }
```

Dynamic Rebinding For Tracing

- 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

```
 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 }
```

Dynamic Rebinding For Tracing

- 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

got_overwriting

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

Road Towards A Real Heap Profiler

- Efficient output data format
  - Consider zstd compression
  - Consider binary data format (e.g. protobuf or similar)
  - Graphing essentially requires a good time series data format
- API for custom allocators
- Analysis GUI
  - FlameGraph visualization is crucial
- Handling other quirks
  - forking, sub processes, clean shutdown, ...

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