OptumSoft Toggle navigation

About

- About OptumSoft
- Fact Sheet
- FAQ
- Thought Leadership
- Executive Team

Legal

- Terms Of Use
- Privacy Policy
- Virtual Patent Marking

Products & Services

Blog

Jobs Jan 5

Contact

Dangers of using dlsym() with RTLD_NEXT

Posted In:

Background

There are times when you want to wrap a library function in order to provide some additional functionality. A common example of this is wrapping the standard library's `malloc()` and `free()` so that you can easily track memory allocations in your program. While there are several techniques for wrapping library functions, one well-known method is using `dlsym()` with `RTLD_NEXT` to locate the *wrapped* function's address so that you can correctly forward calls to it.

Problem

So what can go wrong? Let's look at an example:

LibWrap.h

```
void* memAlloc(size_t s);
// Allocate a memory block of size 's' bytes.
void memDel(void* p);
// Free the block of memory pointed to by 'p'.
```

LibWrap.c

```
#define _GNU_SOURCE
#include <dlfcn.h>
#include "LibWrap.h"

static void* malloc(size_t s) {
   // Wrapper for standard library's 'malloc'.
   // The 'static' keyword forces all calls to malloc() in this file to res
   // to this functions.
   void* (*origMalloc)(size_t) = dlsym(RTLD_NEXT,"malloc");
   return origMalloc(s);
}

static void free(void* p) {
   // Wrapper for standard library's 'free'.
   // The 'static' keyword forces all calls to free() in this file to resol
   // to this functions.
   void (*origFree)(void*) = dlsym(RTLD_NEXT,"free");
   origFree(p);
}

void* memAlloc(size_t s) {
   return malloc(s);
   // Call the malloc() wrapper.
}

void memDel(void* p) {
   free(p);
   // Call the free() wrapper.
}
```

Main.c

```
#include <malloc.h>
#include "LibWrap.h"

int main() {
   struct mallinfo beforeMalloc = mallinfo();
   printf("Bytes allocated before malloc: %d\n",beforeMalloc.uordblks);
```

```
    void* p = memAlloc(57);
    struct mallinfo afterMalloc = mallinfo();
    printf("Bytes allocated after malloc: %d\n",afterMalloc.uordblks);

    memDel(p);
    struct mallinfo afterFree = mallinfo();
    printf("Bytes allocated after free: %d\n",afterFree.uordblks);

    return 0;
}
```

First compile `LibWrap.c` into a shared library:

```
$ gcc -Wall -Werror -fPIC -shared -o libWrap.so LibWrap.c
```

Next compile `Main.c` and link it against the `libWrap.so` that we just created:

```
$ gcc -Wall -Werror -o Main Main.c ./libWrap.so -ldl
```

Time to run the program!

```
$ ./Main
Bytes allocated before malloc: 0
Bytes allocated after malloc: 80
Bytes allocated after free: 0
```

So far, so good. No surprises. We allocated a bunch of memory and then freed it. The statistics returned by `mallinfo()` confirm this.

Out of curiosity, let's look at `ldd` output for the application binary we created.

```
$ ldd Main
       linux-vdso.so.1 =>  (0x00007fff1b1fe000)
       ./libWrap.so (0x00007fe7d2755000)
       libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007fe7d2542000)
       libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fe7d217c000)
       /lib64/ld-linux-x86-64.so.2 (0x00007fe7d2959000)
```

note	The `ldd` output is from Ubuntu 14.04.1 LTS for x86-64. Your output may differ.

Take note of the relative placement of `libWrap.so` with respect to `libc.so.6`: `libWrap.so` comes *before* `libc.so.6`. Remember this. It will be important later.

Now for fun, let's re-compile `Main.c` with `libc.so.6` explicitly specified on the command-line and coming *before* `libWrap.so`:

```
$ gcc -Wall -Werror -o Main Main.c /lib/x86_64-linux-gnu/libc.so.6 ./libWra
```

Re-run:

```
$ ./Main
Bytes allocated before malloc: 0
Bytes allocated after malloc: 80
Bytes allocated after free: 80
```

Uh oh, why are we leaking memory all of a sudden? We de-allocate everything we allocate, so why the memory leak?

It turns out that the leak is occurring because we are *not actually forwarding `malloc()` and `free()` calls to `libc.so.6`'s implementations. Instead, we are forwarding them to `malloc()` and `free()` inside `ld-linux-x86-64.so.2`!*

"What are you talking about?!" you might be asking.

Well, it just so happens that `ld-linux-x86-64.so.2`, which is the dynamic linker/loader, has its own copy of `malloc()` and `free()`. Why? Because `ld-linux` has to allocate memory from the heap *before* it loads `libc.so.6`. But the version of `malloc/free` that `ld-linux` has does not actually free memory!

> See `elf/dl-minimal.c` in glibc source code for `ld-linux`'s `malloc/free` implementation.

But why does `libWrap.so` forward calls to `ld-linux` instead of `libc`? The answer comes down to how `dlsym()` searches for symbols when `RTLD_NEXT` is specified. Here's the relevant excerpt from the `dlsym(3)` man page:

> [RTLD_NEXT] will find the next occurrence of a function in the search order after the current library. This allows one to provide a wrapper around a function in another shared library.

— dlsym(3)

To understand this better, take a look at `ldd` output for the new `Main` binary:

```
$ ldd Main
        linux-vdso.so.1 =>  (0x00007fffe1da0000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f32c2e91000)
        ./libWrap.so (0x00007f32c2c8f000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f32c2a8a000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f32c3267000)
```

Unlike earlier, `libWrap.so` comes *after* `libc.so.6`. So when `dlsym()` is called inside `libWrap.so` to search for functions, it skips `libc.so.6` since it precedes `libWrap.so` in the search order list. That means the searches continue through to `ld-linux-x86-64.so.2` where they find linker/loader's `malloc/free` and return pointers to those functions. And so, `libWrap.so` ends up forwading calls to `ld-linux` instead of `libc`!

> Exercise to the reader: Verify that `malloc/free` calls are getting forwarded to `ld-linux` instead of `libc` by stepping through `Main` with GDB.

At this point you might be wondering: We ran a somewhat funky command to build our application and then encountered a memory leak due to weird library linking order caused by said command. Isn't this whole thing a silly contrived scenario?

The answer is unfortunately no. At OptumSoft, we recently encountered this very same memory leak with a binary compiled using the standard `./configure && make` on x86-64 Ubuntu 14.04.1 LTS. For reasons we don't understand, the linking order for the binary was such that using `dlsym()` with RTLD_NEXT to lookup `malloc/free` resulted in pointers to implementations inside `ld-linux`. It took a ton of effort and invaluable help from Mozilla's rr tool to root-cause the issue. After the whole ordeal, we decided to write a blog post about this strange behavior in case someone else encounters it in the future.

Solution

If you find `dlsym()` with RTLD_NEXT returning pointers to `malloc/free` inside `ld-linux`, what can you do?

For starters, you need to detect that a function address indeed does belong to `ld-linux` using `dladdr()`:

```
void* func = dlsym(RTLD_NEXT,"malloc");
Dl_info dlInfo;
if(!dladdr(func,&dlInfo)) {
   // dladdr() failed.
}
if(strstr(dlInfo.dli_fname,"ld-linux")) {
   // 'malloc' is inside linker/loader.
}
```

Once you have figured out that a function is inside `ld-linux`, you need to decide what to do next. Unfortunately, there is no straightforward way to continue searching for the same function name in all other libraries. But if you know the name of a specific library in which the function exists (e.g. libc), you can use `dlopen()` and `dlsym()` to fetch the desired pointer:

```
void* handle = dlopen("libc.so.6",RTLD_LAZY);
// NOTE: libc.so.6 may *not* exist on Alpha and IA-64 architectures.
if(!handle) {
```

```
   // dlopen() failed.
}
void* func = dlsym(handle,"free");
if(!func) {
   // Bad! 'free' was not found inside libc.
}
```

`dlopen`‘ing a library to replace `malloc/free` is generally frowned upon. Use at your own risk.

Summary

- One can use `dlsym()` with `RTLD_NEXT` to implement wrappers around `malloc()` and `free()`.
- Due to unexpected linking behavior, `dlsym()` when using `RTLD_NEXT` can return pointers to `malloc/free` implementations inside `ld-linux` (dynamic linker/loader). Using `ld-linux`‘s `malloc/free` for general heap allocations leads to memory leaks because that particular version of `free()` doesn’t actually release memory.
- You can check if an address returned by `dlsym()` belongs to `ld-linux` via `dladdr()`. You can also lookup a function in a specific library using `dlopen()` and `dlsym()`.

« The TACC® Difference

Post A Comment

Write your message here...

Your Full Name

E-mail Address

Website

Submit

Search

Start Typing