Load-time relocation of shared libraries (https://eli.thegreenplace.net/2011/08/25/load-time-relocation-of-shared-libraries)

August 25, 2011 at 14:47 **Tags** Assembly (https://eli.thegreenplace.net/tag/assembly) , C & C++ (https://eli.thegreenplace.net/tag/c-c) , Linkers and loaders (https://eli.thegreenplace.net/tag/linkers-and-loaders) , Linux (https://eli.thegreenplace.net/tag/linux)

This article's aim is to explain how a modern operating system makes it possible to use shared libraries with load-time relocation. It focuses on the Linux OS running on 32-bit x86, but the general principles apply to other OSes and CPUs as well.

Note that shared libraries have many names - shared libraries, shared objects, dynamic shared objects (DSOs), dynamically linked libraries (DLLs - if you're coming from a Windows background). For the sake of consistency, I will try to just use the name "shared library" throughout this article.

Loading executables

Linux, similarly to other OSes with virtual memory support, loads executables to a fixed memory address. If we examine the ELF header of some random executable, we'll see an *Entry point address*:

```
$ readelf -h /usr/bin/uptime
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  [...] some header fields
  Entry point address:               0x8048470
  [...] some header fields
```

This is placed by the linker to tell the OS where to start executing the executable's code [1]. And indeed if we then load the executable with GDB and examine the address `0x8048470`, we'll see the first instructions of the executable's `.text` segment there.

What this means is that the linker, when linking the executable, can fully resolve all *internal* symbol references (to functions and data) to fixed and final locations. The linker does some relocations of its own [2], but eventually the output it produces contains no additional relocations.

Or does it? Note that I emphasized the word *internal* in the previous paragraph. As long as the executable needs no shared libraries [3], it needs no relocations. But if it *does* use shared libraries (as do the vast majority of Linux applications), symbols taken from these shared libraries need to be relocated, because of how shared libraries are loaded.

Loading shared libraries

Unlike executables, when shared libraries are being built, the linker can't assume a known load address for their code. The reason for this is simple. Each program can use any number of shared libraries, and there's simply no way to know in advance where any given shared library will be loaded in the process's virtual memory. Many solutions were invented for this problem over the years, but in this article I will just focus on the ones currently used by Linux.

But first, let's briefly examine the problem. Here's some sample C code [4] which I compile into a shared library:

```
int myglob = 42;

int ml_func(int a, int b)
{
    myglob += a;
    return b + myglob;
}
```

Note how `ml_func` references `myglob` a few times. When translated to x86 assembly, this will involve a `mov` instruction to pull the value of `myglob` from its location in memory into a register. `mov` requires an absolute address - so how does the linker know which address to place in it? The answer is - it doesn't. As I mentioned above, shared libraries have no pre-defined load address - it will be decided at runtime.

In Linux, the *dynamic loader* [5] is a piece of code responsible for preparing programs for running. One of its tasks is to load shared libraries from disk into memory, when the running executable requests them. When a shared library is loaded into memory, it is then adjusted for its newly determined load location. It is the job of the dynamic loader to solve the problem presented in the previous paragraph.

There are two main approaches to solve this problem in Linux ELF shared libraries:

1. Load-time relocation
2. Position independent code (PIC)

Although PIC is the more common and nowadays-recommended solution, in this article I will focus on load-time relocation. Eventually I plan to cover both approaches and write a separate article on PIC, and I think starting with load-time relocation will make PIC easier to explain later. (*Update 03.11.2011*: the article about PIC (https://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries/) was published)

Linking the shared library for load-time relocation

To create a shared library that has to be relocated at load-time, I'll compile it without the `-fPIC` flag (which would otherwise trigger PIC generation):

```
gcc -g -c ml_main.c -o ml_mainreloc.o
gcc -shared -o libmlreloc.so ml_mainreloc.o
```

The first interesting thing to see is the entry point of `libmlreloc.so`:

```
$ readelf -h libmlreloc.so
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  [...] some header fields
  Entry point address:               0x3b0
  [...] some header fields
```

For simplicity, the linker just links the shared object for address `0x0` (the `.text` section starting at `0x3b0`), knowing that the loader will move it anyway. Keep this fact in mind - it will be useful later in the article.

Now let's look at the disassembly of the shared library, focusing on `ml_func`:

```
$ objdump -d -Mintel libmlreloc.so

libmlreloc.so:     file format elf32-i386

[...] skipping stuff

0000046c <ml_func>:
 46c: 55                      push   ebp
 46d: 89 e5                   mov    ebp,esp
 46f: a1 00 00 00 00          mov    eax,ds:0x0
 474: 03 45 08                add    eax,DWORD PTR [ebp+0x8]
 477: a3 00 00 00 00          mov    ds:0x0,eax
 47c: a1 00 00 00 00          mov    eax,ds:0x0
 481: 03 45 0c                add    eax,DWORD PTR [ebp+0xc]
 484: 5d                      pop    ebp
 485: c3                      ret

[...] skipping stuff
```

After the first two instructions which are part of the prologue [6], we see the compiled version of `myglob += a` [7]. The value of `myglob` is taken from memory into `eax`, incremented by `a` (which is at `ebp+0x8`) and then placed back into memory.

But wait, the `mov` takes `myglob`? Why? It appears that the actual operand of `mov` is just `0x0` [8]. What gives? This is how relocations work. The linker places some provisional pre-defined value (`0x0` in this case) into the instruction stream, and then creates a special relocation entry pointing to this place. Let's examine the relocation entries for this shared library:

```
$ readelf -r libmlreloc.so

Relocation section '.rel.dyn' at offset 0x2fc contains 7 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00002008  00000008 R_386_RELATIVE
00000470  00000401 R_386_32          0000200C   myglob
00000478  00000401 R_386_32          0000200C   myglob
0000047d  00000401 R_386_32          0000200C   myglob
[...] skipping stuff
```

The `rel.dyn` section of ELF is reserved for dynamic (load-time) relocations, to be consumed by the dynamic loader. There are 3 relocation entries for `myglob` in the section showed above, since there are 3 references to `myglob` in the disassembly. Let's decipher the first one.

It says: go to offset 0x470 in this object (shared library), and apply relocation of type `R_386_32` to it for symbol `myglob`. If we consult the ELF spec we see that relocation type `R_386_32` means: take the value at the offset specified in the entry, add the address of the symbol to it, and place it back into the offset.

What do we have at offset `0x470` in the object? Recall this instruction from the disassembly of `ml_func`:

```
46f:  a1 00 00 00 00          mov    eax,ds:0x0
```

`a1` encodes the `mov` instruction, so its operand starts at the next address which is `0x470`. This is the `0x0` we see in the disassembly. So back to the relocation entry, we now see it says: add the address of `myglob` to the operand of that `mov` instruction. In other words it tells the dynamic loader - once you perform actual address assignment, put the real address of `myglob` into `0x470`, thus replacing the operand of `mov` by the correct symbol value. Neat, huh?

Note also the "Sym. value" column in the relocation section, which contains `0x200C` for `myglob`. This is the offset of `myglob` in the virtual memory image of the shared library (which, recall, the linker assumes is just loaded at `0x0`). This value can also be examined by looking at the symbol table of the library, for example with `nm`:

```
$ nm libmlreloc.so
[...] skipping stuff
0000200c D myglob
```

This output also provides the offset of `myglob` inside the library. `D` means the symbol is in the initialized data section (`.data`).

Load-time relocation in action

To see the load-time relocation in action, I will use our shared library from a simple driver executable. When running this executable, the OS will load the shared library and relocate it appropriately.

Curiously, due to the address space layout randomization feature (http://en.wikipedia.org/wiki/Address_space_layout_randomization) which is enabled in Linux, relocation is relatively difficult to follow, because every time I run the executable, the `libmlreloc.so` shared library gets placed in a different virtual memory address [9].

This is a rather weak deterrent, however. There is a way to make sense in it all. But first, let's talk about the segments our shared library consists of:

```
$ readelf --segments libmlreloc.so

Elf file type is DYN (Shared object file)
Entry point 0x3b0
There are 6 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000000 0x00000000 0x00000000 0x004e8 0x004e8 R E 0x1000
  LOAD           0x000f04 0x00001f04 0x00001f04 0x0010c 0x00114 RW  0x1000
  DYNAMIC        0x000f18 0x00001f18 0x00001f18 0x000d0 0x000d0 RW  0x4
  NOTE           0x0000f4 0x000000f4 0x000000f4 0x00024 0x00024 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
  GNU_RELRO      0x000f04 0x00001f04 0x00001f04 0x000fc 0x000fc R   0x1

 Section to Segment mapping:
  Segment Sections...
   00     .note.gnu.build-id .hash .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .re
   01     .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
   02     .dynamic
   03     .note.gnu.build-id
   04
   05     .ctors .dtors .jcr .dynamic .got
```

To follow the `myglob` symbol, we're interested in the second segment listed here. Note a couple of things:

- In the section to segment mapping in the bottom, segment 01 is said to contain the `.data` section, which is the home of `myglob`
- The `VirtAddr` column specifies that the second segment starts at `0x1f04` and has size `0x10c`, meaning that it extends until `0x2010` and thus contains `myglob` which is at `0x200C`.

Now let's use a nice tool Linux gives us to examine the load-time linking process - the dl_iterate_phdr function (http://linux.die.net/man/3/dl_iterate_phdr), which allows an application to inquire at runtime which shared libraries it has loaded, and more importantly - take a peek at their program headers.

So I'm going to write the following code into `driver.c`:

```
#define _GNU_SOURCE
#include <link.h>
#include <stdlib.h>
#include <stdio.h>


static int header_handler(struct dl_phdr_info* info, size_t size, void* data)
{
    printf("name=%s (%d segments) address=%p\n",
            info->dlpi_name, info->dlpi_phnum, (void*)info->dlpi_addr);
    for (int j = 0; j < info->dlpi_phnum; j++) {
         printf("\t\t header %2d: address=%10p\n", j,
             (void*) (info->dlpi_addr + info->dlpi_phdr[j].p_vaddr));
         printf("\t\t\t type=%u, flags=0x%X\n",
                 info->dlpi_phdr[j].p_type, info->dlpi_phdr[j].p_flags);
    }
    printf("\n");
    return 0;
}


extern int ml_func(int, int);


int main(int argc, const char* argv[])
{
    dl_iterate_phdr(header_handler, NULL);

    int t = ml_func(argc, argc);
    return t;
}
```

`header_handler` implements the callback for `dl_iterate_phdr`. It will get called for all libraries and report their names and load addresses, along with all their segments. It also invokes `ml_func`, which is taken from the `libmlreloc.so` shared library.

To compile and link this driver with our shared library, run:

```
gcc -g -c driver.c -o driver.o
gcc -o driver driver.o -L. -lmlreloc
```

Running the driver stand-alone we get the information, but for each run the addresses are different. So what I'm going to do is run it under *gdb* [10], see what it says, and then use *gdb* to further query the process's memory space:

```
$ gdb -q driver
Reading symbols from driver...done.
(gdb) b driver.c:31
Breakpoint 1 at 0x804869e: file driver.c, line 31.
(gdb) r
Starting program: driver
[...] skipping output
name=./libmlreloc.so (6 segments) address=0x12e000
               header  0: address=  0x12e000
                       type=1, flags=0x5
               header  1: address=  0x12ff04
                       type=1, flags=0x6
               header  2: address=  0x12ff18
                       type=2, flags=0x6
               header  3: address=  0x12e0f4
                       type=4, flags=0x4
               header  4: address=  0x12e000
                       type=1685382481, flags=0x6
               header  5: address=  0x12ff04
                       type=1685382482, flags=0x4

[...] skipping output
Breakpoint 1, main (argc=1, argv=0xbffff3d4) at driver.c:31
31    }
(gdb)
```

Since `driver` reports all the libraries it loads (even implicitly, like `libc` or the dynamic loader itself), the output is lengthy and I will just focus on the report about `libmlreloc.so`. Note that the 6 segments are the same segments reported by `readelf`, but this time relocated into their final memory locations.

Let's do some math. The output says `libmlreloc.so` was placed in virtual address `0x12e000`. We're interested in the second segment, which as we've seen in `readelf` is at ofset `0x1f04`. Indeed, we see in the output it was loaded to address `0x12ff04`. And since `myglob` is at offset `0x200c` in the file, we'd expect it to now be at address `0x13000c`.

So, let's ask GDB:

```
(gdb) p &myglob
$1 = (int *) 0x13000c
```

Excellent! But what about the code of `ml_func` which refers to `myglob`? Let's ask GDB again:

```
(gdb) set disassembly-flavor intel
(gdb) disas ml_func
Dump of assembler code for function ml_func:
   0x0012e46c <+0>:   push   ebp
   0x0012e46d <+1>:   mov    ebp,esp
   0x0012e46f <+3>:   mov    eax,ds:0x13000c
   0x0012e474 <+8>:   add    eax,DWORD PTR [ebp+0x8]
   0x0012e477 <+11>:  mov    ds:0x13000c,eax
   0x0012e47c <+16>:  mov    eax,ds:0x13000c
   0x0012e481 <+21>:  add    eax,DWORD PTR [ebp+0xc]
   0x0012e484 <+24>:  pop    ebp
   0x0012e485 <+25>:  ret
End of assembler dump.
```

As expected, the real address of `myglob` was placed in all the `mov` instructions referring to it, just as the relocation entries specified.

Relocating function calls

So far this article demonstrated relocation of data references - using the global variable `myglob` as an example. Another thing that needs to be relocated is code references - in other words, function calls. This section is a brief guide on how this gets done. The pace is much faster than in the rest of this article, since I can now assume the reader understands what relocation is all about.

Without further ado, let's get to it. I've modified the code of the shared library to be the following:

```
int myglob = 42;

int ml_util_func(int a)
{
    return a + 1;
}

int ml_func(int a, int b)
{
    int c = b + ml_util_func(a);
    myglob += c;
    return b + myglob;
}
```

`ml_util_func` was added and it's being used by `ml_func`. Here's the disassembly of `ml_func` in the linked shared library:

```
000004a7 <ml_func>:
 4a7:   55                      push   ebp
 4a8:   89 e5                   mov    ebp,esp
 4aa:   83 ec 14                sub    esp,0x14
 4ad:   8b 45 08                mov    eax,DWORD PTR [ebp+0x8]
 4b0:   89 04 24                mov    DWORD PTR [esp],eax
 4b3:   e8 fc ff ff ff          call   4b4 <ml_func+0xd>
 4b8:   03 45 0c                add    eax,DWORD PTR [ebp+0xc]
 4bb:   89 45 fc                mov    DWORD PTR [ebp-0x4],eax
 4be:   a1 00 00 00 00          mov    eax,ds:0x0
 4c3:   03 45 fc                add    eax,DWORD PTR [ebp-0x4]
 4c6:   a3 00 00 00 00          mov    ds:0x0,eax
 4cb:   a1 00 00 00 00          mov    eax,ds:0x0
 4d0:   03 45 0c                add    eax,DWORD PTR [ebp+0xc]
 4d3:   c9                      leave
 4d4:   c3                      ret
```

What's interesting here is the instruction at address `0x4b3` - it's the call to `ml_util_func`. Let's dissect it:

`e8` is the opcode for `call`. The argument of this `call` is the offset relative to the next instruction. In the disassembly above, this argument is `0xfffffffc`, or simply `-4`. So the `call` currently points to itself. This clearly isn't right - but let's not forget about relocation. Here's what the relocation section of the shared library looks like now:

```
$ readelf -r libmlreloc.so

Relocation section '.rel.dyn' at offset 0x324 contains 8 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00002008  00000008 R_386_RELATIVE
000004b4  00000502 R_386_PC32        0000049c   ml_util_func
000004bf  00000401 R_386_32          0000200c   myglob
000004c7  00000401 R_386_32          0000200c   myglob
000004cc  00000401 R_386_32          0000200c   myglob
[...] skipping stuff
```

If we compare it to the previous invocation of `readelf -r`, we'll notice a new entry added for `ml_util_func`. This entry points at address `0x4b4` which is the argument of the `call` instruction, and its type is `R_386_PC32`. This relocation type is more complicated than `R_386_32`, but not by much.

It means the following: take the value at the offset specified in the entry, add the address of the symbol to it, subtract the address of the offset itself, and place it back into the word at the offset. Recall that this relocation is done at *load-time*, when the final load addresses of the symbol and the relocated offset itself are already known. These final addresses participate in the computation.

What does this do? Basically, it's a *relative* relocation, taking its location into account and thus suitable for arguments of instructions with relative addressing (which the `e8 call` is). I promise it will become clearer once we get to the real numbers.

I'm now going to build the driver code and run it under GDB again, to see this relocation in action. Here's the GDB session, followed by explanations:

```
 $ gdb -q driver
 Reading symbols from driver...done.
 (gdb) b driver.c:31
 Breakpoint 1 at 0x804869e: file driver.c, line 31.
 (gdb) r
 Starting program: driver
 [...] skipping output
 name=./libmlreloc.so (6 segments) address=0x12e000
              header  0: address=  0x12e000
                      type=1, flags=0x5
              header  1: address=  0x12ff04
                      type=1, flags=0x6
              header  2: address=  0x12ff18
                      type=2, flags=0x6
              header  3: address=  0x12e0f4
                      type=4, flags=0x4
              header  4: address=  0x12e000
                      type=1685382481, flags=0x6
              header  5: address=  0x12ff04
                      type=1685382482, flags=0x4

[...] skipping output
Breakpoint 1, main (argc=1, argv=0xbffff3d4) at driver.c:31
31    }
(gdb)  set disassembly-flavor intel
(gdb) disas ml_util_func
Dump of assembler code for function ml_util_func:
   0x0012e49c <+0>:   push   ebp
   0x0012e49d <+1>:   mov    ebp,esp
   0x0012e49f <+3>:   mov    eax,DWORD PTR [ebp+0x8]
   0x0012e4a2 <+6>:   add    eax,0x1
   0x0012e4a5 <+9>:   pop    ebp
   0x0012e4a6 <+10>:  ret
End of assembler dump.
(gdb) disas /r ml_func
Dump of assembler code for function ml_func:
   0x0012e4a7 <+0>:    55     push   ebp
   0x0012e4a8 <+1>:    89 e5  mov    ebp,esp
   0x0012e4aa <+3>:    83 ec 14       sub    esp,0x14
   0x0012e4ad <+6>:    8b 45 08       mov    eax,DWORD PTR [ebp+0x8]
   0x0012e4b0 <+9>:    89 04 24       mov    DWORD PTR [esp],eax
   0x0012e4b3 <+12>:   e8 e4 ff ff ff call   0x12e49c <ml_util_func>
   0x0012e4b8 <+17>:   03 45 0c       add    eax,DWORD PTR [ebp+0xc]
   0x0012e4bb <+20>:   89 45 fc       mov    DWORD PTR [ebp-0x4],eax
   0x0012e4be <+23>:   a1 0c 00 13 00 mov    eax,ds:0x13000c
   0x0012e4c3 <+28>:   03 45 fc       add    eax,DWORD PTR [ebp-0x4]
   0x0012e4c6 <+31>:   a3 0c 00 13 00 mov    ds:0x13000c,eax
   0x0012e4cb <+36>:   a1 0c 00 13 00 mov    eax,ds:0x13000c
   0x0012e4d0 <+41>:   03 45 0c       add    eax,DWORD PTR [ebp+0xc]
   0x0012e4d3 <+44>:   c9     leave
   0x0012e4d4 <+45>:   c3     ret
```

```
End of assembler dump.
(gdb)
```

The important parts here are:

1. In the printout from `driver` we see that the first segment (the code segment) of `libmlreloc.so` has been mapped to `0x12e000` [11]
2. `ml_util_func` was loaded to address `0x0012e49c`
3. The address of the relocated offset is `0x0012e4b4`
4. The call in `ml_func` to `ml_util_func` was patched to place `0xffffffe4` in the argument (I disassembled `ml_func` with the `/r` flag to show raw hex in addition to disassembly), which is interpreted as the correct offset to `ml_util_func`.

Obviously we're most interested in how (4) was done. Again, it's time for some math. Interpreting the `R_386_PC32` relocation entry mentioned above, we have:

Take the value at the offset specified in the entry (`0xfffffffc`), add the address of the symbol to it (`0x0012e49c`), subtract the address of the offset itself (`0x0012e4b4`), and place it back into the word at the offset. Everything is done assuming 32-bit 2-s complement, of course. The result is `0xffffffe4`, as expected.

Extra credit: Why was the call relocation needed?

This is a "bonus" section that discusses some peculiarities of the implementation of shared library loading in Linux. If all you wanted was to understand how relocations are done, you can safely skip it.

When trying to understand the call relocation of `ml_util_func`, I must admit I scratched my head for some time. Recall that the argument of `call` is a *relative offset*. Surely the offset between the `call` and `ml_util_func` itself doesn't change when the library is loaded - they both are in the code segment which gets moved as one whole chunk. So why is the relocation needed at all?

Here's a small experiment to try: go back to the code of the shared library, add `static` to the declaration of `ml_util_func`. Re-compile and look at the output of `readelf -r` again.

Done? Anyway, I will reveal the outcome - the relocation is gone! Examine the disassembly of `ml_func` - there's now a correct offset placed as the argument of `call` - no relocation required. What's going on?

When tying global symbol references to their actual definitions, the dynamic loader has some rules about the order in which shared libraries are searched. The user can also influence this order by setting the `LD_PRELOAD` environment variable.

There are too many details to cover here, so if you're really interested you'll have to take a look at the ELF standard, the dynamic loader man page and do some Googling. In short, however, when `ml_util_func` is global, it may be overridden in the executable or another shared library, so when linking our shared library, the linker can't just assume the offset is known and hard-code it [12]. It makes all references to global symbols relocatable in order to allow the dynamic loader to decide how to resolve them. This is why declaring the function `static` makes a difference - since it's no longer global or exported, the linker can hard-code its offset in the code.

Extra credit #2: Referencing shared library data from the executable

Again, this is a bonus section that discusses an advanced topic. It can be skipped safely if you're tired of this stuff.

In the example above, `myglob` was only used internally in the shared library. What happens if we reference it from the program (`driver.c`)? After all, `myglob` is a global variable and thus visible externally.

Let's modify `driver.c` to the following (note I've removed the segment iteration code):

```
#include <stdio.h>

extern int ml_func(int, int);
extern int myglob;

int main(int argc, const char* argv[])
{
    printf("addr myglob = %p\n", (void*)&myglob);
    int t = ml_func(argc, argc);
    return t;
}
```

It now prints the address of `myglob`. The output is:

```
addr myglob = 0x804a018
```

Wait, something doesn't compute here. Isn't `myglob` in the shared library's address space? `0x804xxxx` looks like the program's address space. What's going on?

Recall that the program/executable is not relocatable, and thus its data addresses have to bound at link time. Therefore, the linker has to create a copy of the variable in the program's address space, and the dynamic loader will use *that* as the relocation address. This is similar to the discussion in the previous section - in a sense, `myglob` in the main program overrides the one in the shared library, and according to the global symbol lookup rules, it's being used instead. If we examine `ml_func` in GDB, we'll see the correct reference made to `myglob`:

```
0x0012e48e <+23>:      a1 18 a0 04 08 mov    eax,ds:0x804a018
```

This makes sense because a `R_386_32` relocation for `myglob` still exists in `libmlreloc.so`, and the dynamic loader makes it point to the correct place where `myglob` now lives.

This is all great, but something is missing. `myglob` is initialized in the shared library (to 42) - how does this initialization value get to the address space of the program? It turns out there's a special relocation entry that the linker builds into the *program* (so far we've only been examining relocation entries in the shared library):

```
$ readelf -r driver

Relocation section '.rel.dyn' at offset 0x3c0 contains 2 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
08049ff0  00000206 R_386_GLOB_DAT    00000000   __gmon_start__
0804a018  00000605 R_386_COPY        0804a018   myglob
[...] skipping stuff
```

Note the `R_386_COPY` relocation for `myglob`. It simply means: copy the value from the symbol's address into this offset. The dynamic loader performs this when it loads the shared library. How does it know how much to copy? The symbol table section contains the size of each symbol; for example the size for `myglob` in the `.symtab` section of `libmlreloc.so` is 4.

I think this is a pretty cool example that shows how the process of executable linking and loading is orchestrated together. The linker puts special instructions in the output for the dynamic loader to consume and execute.

Conclusion

Load-time relocation is one of the methods used in Linux (and other OSes) to resolve internal data and code references in shared libraries when loading them into memory. These days, position independent code (PIC) is a more popular approach, and some modern systems (such as x86-64) no longer support load-time relocation.

Still, I decided to write an article on load-time relocation for two reasons. First, load-time relocation has a couple of advantages over PIC on some systems, especially in terms of performance. Second, load-time relocation is IMHO simpler to understand without prior knowledge, which will make PIC easier to explain in the future. (*Update 03.11.2011*: the article about PIC (https://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries/) was published)

Regardless of the motivation, I hope this article has helped to shed some light on the magic going behind the scenes of linking and loading shared libraries in a modern OS.

[1] For some more information about this entry point, see the section "Digression – process addresses and entry point" of this article (https://eli.thegreenplace.net/2011/01/27/how-debuggers-work-part-2-breakpoints/).

[2] *Link-time relocation* happens in the process of combining multiple object files into an executable (or shared library). It involves quite a lot of relocations to resolve symbol references between the object files. Link-time relocation is a more complex topic than load-time relocation, and I won't cover it in this article.

[3] This can be made possible by compiling all your libraries into static libraries (with `ar` combining object files instead `gcc -shared`), and providing the `-static` flag to `gcc` when linking the executable - to avoid linkage with the shared version of `libc`.

[4] `ml` simply stands for "my library". Also, the code itself is absolutely non-sensical and only used for purposes of demonstration.

[5] Also called "dynamic linker". It's a shared object itself (though it can also run as an executable), residing at `/lib/ld-linux.so.2` (the last number is the SO version and may be different).

[6] If you're not familiar with how x86 structures its stack frames, this would be a good time to read this article (https://eli.thegreenplace.net/2011/02/04/where-the-top-of-the-stack-is-on-x86/).

[7] You can provide the `-l` flag to `objdump` to add C source lines into the disassembly, making it clearer what gets compiled to what. I've omitted it here to make the output shorter.

[8] I'm looking at the left-hand side of the output of `objdump`, where the raw memory bytes are. `a1 00 00 00 00` means `mov` to `eax` with operand `0x0`, which is interpreted by the disassembler as `ds:0x0`.

[9] So `ldd` invoked on the executable will report a different load address for the shared library each time it's run.

[10] Experienced readers will probably note that I could ask GDB about `i shared` to get the load-address of the shared library. However, `i shared` only mentions the load location of the whole library (or, even more accurately, its entry point), and I was interested in the segments.

[11] What, `0x12e000` again? Didn't I just talk about load-address randomization? It turns out the dynamic loader can be manipulated to turn this off, for purposes of debugging. This is exactly what GDB is doing.

[12] Unless it's passed the `-Bsymbolic` flag. Read all about it in the man page of `ld`.

For comments, please send me an email (mailto:eliben@gmail.com).

Back to top