Dynamic Linker

Sooner or later you'll reach the point where you want shared libraries. Here we won't discuss the difference between static and dynamic linking, you should be already familiar with that.

This article is about ELF on x86_64 architecture, but can be easily adopted to other systems as the concepts are similar.

Contents

Home work

Memory Layout

I suppose you have created a new, empty address space, and you've already loaded the executable in it. I also assume that you've loaded the libc shared library after the executable's segment. And now you're stuck, because you don't know how to call printf from the executable.

As your executable was loaded by you, you should know the virtual address of ELF magic bytes in the memory. Use that for start.

```
Elf64_Ehdr *ehdr = (Elf64_Ehdr *)(ptr);

if(!kmemcmp(ehdr->e_ident, ELFMAG,SELFMAG) &&
   ehdr->e_ident[EI_CLASS] == ELFCLASS64 &&
   ehdr->e_ident[EI_DATA] == ELFDATA2LSB &&
   ehdr->e_type == ET_EXEC && ehdr->e_shnum > 0)
{
   // We have a valid image with sections
}
```

First we check the magic bytes and the format of the ELF64. As an extra, we also check whether it's executable and has a non-empty section table (we'll going to need it).

Now let's see what the GNU toolchain does for us to find printf.

Segment Local Calls

In order to figure that out, first we should know how a segment local call works. For that we'll use a very minimal source.

```
void localfunction()
{
}

int main(int,char**)
{
    localfunction();
}
```

That compiles to:

```
$ objdump -d test
000000000020016f <localfunction>:
  20016f:   55                      push   %rbp
  200170:   48 89 e5                mov    %rsp,%rbp
  200173:   90                      nop
  200174:   5d                      pop    %rbp
  200175:   c3                      retq

0000000000200176 <main>:
  200176:   55                      push   %rbp
  200177:   48 89 e5                mov    %rsp,%rbp
  20017a:   b8 00 00 00 00          mov    $0x0,%eax
  20017f:   e8 eb ff ff ff          callq  20016f <localfunction>
  200184:   90                      nop
  200185:   5d                      pop    %rbp
  200186:   c3                      retq
```

That's trivial, a rip relative addressing is used at 20017f.

Inter-segment Calls

Now let's modify the source a bit to use a libc call:

```
int main(int,char**)
{
    printf("Hello World");
}
```

Compile and see what's generated.

```
000000000020016f <main>:
  20016f:   55                      push   %rbp
  200170:   48 89 e5                mov    %rsp,%rbp
  200173:   48 8d 3d b6 01 00 00    lea    0x1b6(%rip),%rdi        # 200330 <_DYNAMIC+0x110>
  20017a:   b8 00 00 00 00          mov    $0x0,%eax
  20017f:   e8 8c 00 00 00          callq  200210 <printf@plt>
  200184:   90                      nop
  200185:   5d                      pop    %rbp
  200186:   c3                      retq

0000000000200200 <printf@plt-0x10>:
  200200:   ff 35 4a 0e 00 00       pushq  0xe4a(%rip)        # 201050 <_GLOBAL_OFFSET_TABLE_+0x8>
  200206:   ff 25 4c 0e 00 00       jmpq   *0xe4c(%rip)        # 201058 <_GLOBAL_OFFSET_TABLE_+0x10>
  20020c:   0f 1f 40 00             nopl   0x0(%rax)
```

```
0000000000200210 <printf@plt>:
  200210:   ff 25 4a 0e 00 00       jmpq   *0xe4a(%rip)        # 201060 <_GLOBAL_OFFSET_TABLE_+0x18>
  200216:   68 00 00 00 00          pushq  $0x0
  20021b:   e9 e0 ff ff ff          jmpq   200200 <main+0x91>
```

What? Two more local functions? What happened here? The GNU toolchain has a concept for lazy run-time linking. That means the address is not resolved until it's referenced. To achieve that, it needs helper functions (generated to the .plt section in the text segment).

When the CPU executes this code, it will first call a normal local function at 20017f. That function is one of the helpers and it's purpose is to load an address from GOT+0x18 and jump to it. By default, the value points to the next instruction, which saves 0 to stack and calls the other helper function at 200200. That one is the reference resolver, and it's job is the replace the address in GOT with a relocated address.

Because the resolver function is not known at link time, it's address is also in GOT at 0x10. What's more it can receive one argument, stored at GOT+0x8. We can also spot that at 200206 the instruction is a jump and not a call, so resolver never returns to this helper, instead it should jump to the relocated address.

Implementing a dynamic linker

All that disassembly teach us two things about the dynamic linker:

```
1. it has to locate and write the GOT
2. it has two parts: load time linker and a run time resolver.
```

The first part runs before the thread is started and saves the second part's address and argument into GOT. On the other hand, the second part runs when the thread is already running, and saves relocated addresses into GOT. As you can see, both parts require the address of GOT. To save resources, one should not locate the GOT twice: this is where the resolver's argument came in. While you can technically insert anything into this slot in the GOT, your best bet will be to store something that will easily identify the requesting program — for instance, the address of the start of the GOT.

To proceed we'll have to locate the GOT in memory and figure out what entries it has.

Locating the GOT

Time to peek on what's in the object file.

```
$ readelf -a test
Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [10] '.got.plt'        PROGBITS         0000000000201048  00001048
       0000000000000020  0000000000000008  WA       0     0     8

Symbol table .symtab contains 23 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
    15: 0000000000201048     0 OBJECT  LOCAL  DEFAULT   10 _GLOBAL_OFFSET_TABLE_
```

Symbol table shows that the GOT is at 201048. We can also see the same value in the section headers at '.got.plt'. That means we don't have to resolve symbols in order to get GOT's address which simplifies the first part. We can also learn that the GOT is 32 (0x20) bytes long in our example.

Note: the address of the GOT can also be found in the dynamic section for most executable files, under an entry tagged 'DT_PLTGOT.'

What's in the GOT?

We already know that

```
1. GOT+0x0 entry is unused
2. GOT+0x8 is an argument to second part
3. GOT+0x10 is function reference to second part
```

But what about the rest, starting at 201060 in our example? Here we have only one reference so it's obvious, but what if we have more references? How should we know which symbol is associated to which entry?

```
$ readelf -a test
Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 4] '.rela.plt'       RELA             00000000002001e8  000001e8
       0000000000000018  0000000000000018  AI       2    10     8

Relocation section .rela.plt at offset 0x1e8 contains 1 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000201060  000100000007 R_X86_64_JUMP_SLO 0000000000000000 printf + 0
```

How convenient that another table is also recorded in the section headers. It's called '.rela.plt' and describes exactly that. When the resolver receives control, the second parameter on the stack will be an index into this section, which is the offset (in 16-byte entries on x86-64, or 8-byte entries on 32-bit x86) of an entry like this that points to the symbol being requested.

Where are my libraries?

So far we assumed that shared libraries are already loaded. It's the case with libc, but how do we know what other shared libraries the executable wants?

```
$ readelf -a test
Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 6] '.dynamic'        DYNAMIC          0000000000200220  00000220
       0000000000000110  0000000000000010  WA       3     0     8

Dynamic section at offset 0x220 contains 12 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libc.so]
```

Not surprising that the answer lies in the section header again. There's a table pointer called '.dynamic'. That table has several records, but what we really are interested in is the ones marked by "NEEDED".

Symbol look up

To find out printf's address we should locate it's symbol first in the shared library.

```
$ readelf -a libc.so
Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 2] '.dynsym'         DYNSYM           0000000100000218  00000218
       00000000000003c0  0000000000000018   A       3     1     8
  [ 3] '.dynstr'         STRTAB           00000001000005d8  000005d8
       0000000000000124  0000000000000000   A       0     0     1

Symbol table .dynsym contains 40 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     8: 0000000100000175    93 FUNC    GLOBAL DEFAULT    1 printf
```

Bingo! It is 100000175 in our example.

Gimme code!

I've put all the above together in a very simple example, see elftool.c (https://gitlab.com/bztsrc/osz/-/blob/master/tools/elftool.c) on gitlab.

When I run it on the executable it gives:

```
$ gcc elftool.c -o elftool
$ ./elftool -d mytestelf.o
Stringtable 000003f0 (118 bytes), symbols 000002b8 (312 bytes, one entry 24)

--- IMPORT ---
Dynamic 00001e20 (464 bytes, one entry 16):
  0. /lib/libc.so.6

GOT 00002000 (104 bytes), Rela 000004d0 (240 bytes, one entry 24):
  0. 00602018 +0 puts
  1. 00602020 +0 fread
  2. 00602028 +0 fclose
  3. 00602030 +0 printf
  4. 00602038 +0 strcmp
  5. 00602040 +0 ftell
  6. 00602048 +0 malloc
  7. 00602050 +0 fseek
  8. 00602058 +0 fopen
  9. 00602060 +0 exit

--- EXPORT ---
```

As you can see here we have an import section, but nothing to be exported. Now let's see a shared library!

```
$ ./elftool -d /lib/libc.so.6
Stringtable 00011038 (23041 bytes), symbols 00003d90 (53928 bytes, one entry 24)

--- IMPORT ---
Dynamic 00197b60 (496 bytes, one entry 16):
  0. /lib/ld-linux-x86-64.so.2

GOT 00198000 (88 bytes), Rela 0001f760 (192 bytes, one entry 24):
```

```
  0. 00398050 +844d0
  1. 00398048 +a8560
  2. 00398040 +7ff80
  3. 00398038 +867c0
  4. 00398030 +823f0
  5. 00398028 +82780
  6. 00398020 +a8640
  7. 00398018 +83b50

--- EXPORT ---
  8. 0008e850 __strspn_c1
  9. 0006aad0 putwchar
 10. 000f8640 __gethostname_chk
 11. 0008e870 __strspn_c2
 12. 0010f210 setrpcent
 13. 0009eda0 __wcstod_l
 14. 0008e8a0 __strspn_c3
 15. 000e8d10 epoll_create
 16. 000d1b50 sched_get_priority_min
 17. 000f8660 __getdomainname_chk
 18. 000e8f20 klogctl
 19. 0002c380 __tolower_l
 20. 0004f440 dprintf
 21. 000b8e00 setuid
 22. 000a3d20 __wcscoll_l
... lot more lines to come ...
```

This time it has hell a lot of functions to export, and also it imports the dynamic linker of Linux with addend offsets in the GOT.

Summary

To summarize a dynamic linker should be look like:

```
1. load-time linker
  1.1. locates GOT by '.got.plt' section, and saves that address in GOT+0x8
  1.2. stores second part's address at GOT+0x10
  1.3. reads '.dynamic' section to load shared libraries
2. run-time reference resolver
  2.1. it's called by a helper
  2.2. that helper places index-0x18 and the address of GOT as arguments on the stack
  2.3. it has to locate '.rela.plt' to get the symbol for the reference
  2.4. the symbol is looked up in the shared library's '.dynsym' section to get relocated address
  2.5. that relocated address has to be saved in the GOT (index and base on the stack)
  2.6. clean up stack, restore registers and jump to the relocated address
```

That's all, hope it helps somebody! Good luck with implementing your own dynamic linker!