Position Independent Code (PIC) in shared libraries (https://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries)

November 03, 2011 at 06:14 **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)

I've described the need for special handling of shared libraries while loading them into the process's address space in a previous article (https://eli.thegreenplace.net/2011/08/25/load-time-relocation-of-shared-libraries/). Briefly, when the linker creates a shared library, it doesn't know in advance where it might be loaded. This creates a problem for the data and code references within the library, which should be somehow made to point to the correct memory locations.

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

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

Load-time relocation was already covered (https://eli.thegreenplace.net/2011/08/25/load-time-relocation-of-shared-libraries/). Here, I want to explain the second approach - PIC.

I originally planned to focus on both x86 and x64 (a.k.a. x86-64) in this article, but as it grew longer and longer I decided it won't be practical. So, it will explain only how PIC works on x86, picking this older architecture specifically because (unlike x64) it wasn't designed with PIC in mind, so implementing PIC on it is a bit trickier. A future (hopefully much shorter) article will build upon the foundation of this one to explain how PIC is implemented on x64.

The code for this post is available online (https://github.com/eliben/code-for-blog/tree/main/2011/pic).

Some problems of load-time relocation

As we've seen in the previous article, load-time relocation is a fairly straightforward method, and it works. PIC, however, is much more popular nowadays, and is usually the recommended method of building shared libraries. Why is this so?

Load-time relocation has a couple of problems: it takes time to perform, and it makes the text section of the library non-shareable.

First, the performance problem. If a shared library was linked with load-time relocation entries, it will take some time to actually perform these relocations when the application is loaded. You may think that the cost shouldn't be too large - after all, the loader doesn't have to scan through the whole text section - it should only look at the relocation entries. But if a complex piece of software loads multiple large shared libraries at start-up, and each shared library must first have its load-time relocations applied, these costs can build up and result in a noticeable delay in the start-up time of the application.

Second, the non-shareable text section problem, which is somewhat more serious. One of the main points of having shared libraries in the first place, is saving RAM. Some common shared libraries are used by multiple applications. If the text section (where the code is) of the shared library can only be loaded into memory once (and then mapped into the virtual memories of many processes), considerable amounts of RAM can be saved. But this is not possible with load-time relocation, since when using this technique the text section has to be modified at load-time to apply the relocations. Therefore, for each application that loaded this shared library, it will have to be wholly placed in RAM again [1]. Different applications won't be able to really share it.

Moreover, having a writable text section (it must be kept writable, to allow the dynamic loader to perform the relocations) poses a security risk, making it easier to exploit the application.

As we'll see in this article, PIC mostly mitigates these problems.

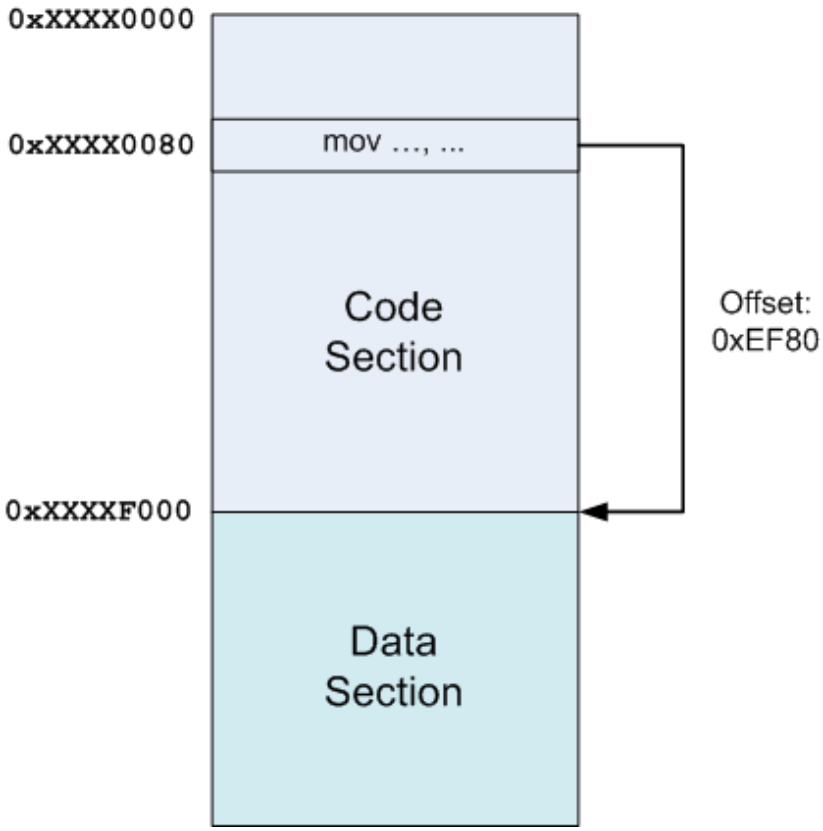
PIC - introduction

The idea behind PIC is simple - add an additional level of indirection to all global data and function references in the code. By cleverly utilizing some artifacts of the linking and loading processes, it's possible to make the text section of the shared library truly *position independent*, in the sense that it can be easily mapped into different memory addresses without needing to change one bit. In the next few sections I will explain in detail how this feat is achieved.

Key insight #1 - offset between text and data sections

One of the key insights on which PIC relies is the offset between the text and data sections, known to the linker *at link-time*. When the linker combines several object files together, it collects their sections (for example, all text sections get unified into a single large text section). Therefore, the linker knows both about the sizes of the sections and about their relative locations.

For example, the text section may be immediately followed by the data section, so the offset from any given instruction in the text section to the beginning of the data section is just the size of the text section minus the offset of the instruction from the beginning of the text section - and both these quantities are known to the linker.



In the diagram above, the code section was loaded into some address (unknown at link-time) 0xXXXX0000 (the X-es literally mean "don't care"), and the data section right after it at offset 0xXXXXF000. Then, if some instruction at offset 0x80 in the code section wants to reference stuff in the data section, the linker knows the relative offset (0xEF80 in this case) and can encode it in the instruction.

Note that it wouldn't matter if another section was placed between the code and data sections, or if the data section preceded the code section. Since the linker knows the sizes of all sections and decides where to place them, the insight holds.

Key insight #2 - making an IP-relative offset work on x86

The above is only useful if we can actually put the relative offset to work. But data references (i.e. in the `mov` instruction) on x86 require absolute addresses. So, what can we do?

If we have a relative address and need an absolute address, what's missing is the value of the instruction pointer (since, by definition, the *relative* address is relative to the instruction's location). There's no instruction to obtain the value of the instruction pointer on x86, but we can use a simple trick to get it. Here's some assembly pseudo-code that demonstrates it:

```
    call TMPLABEL
TMPLABEL:
    pop ebx
```

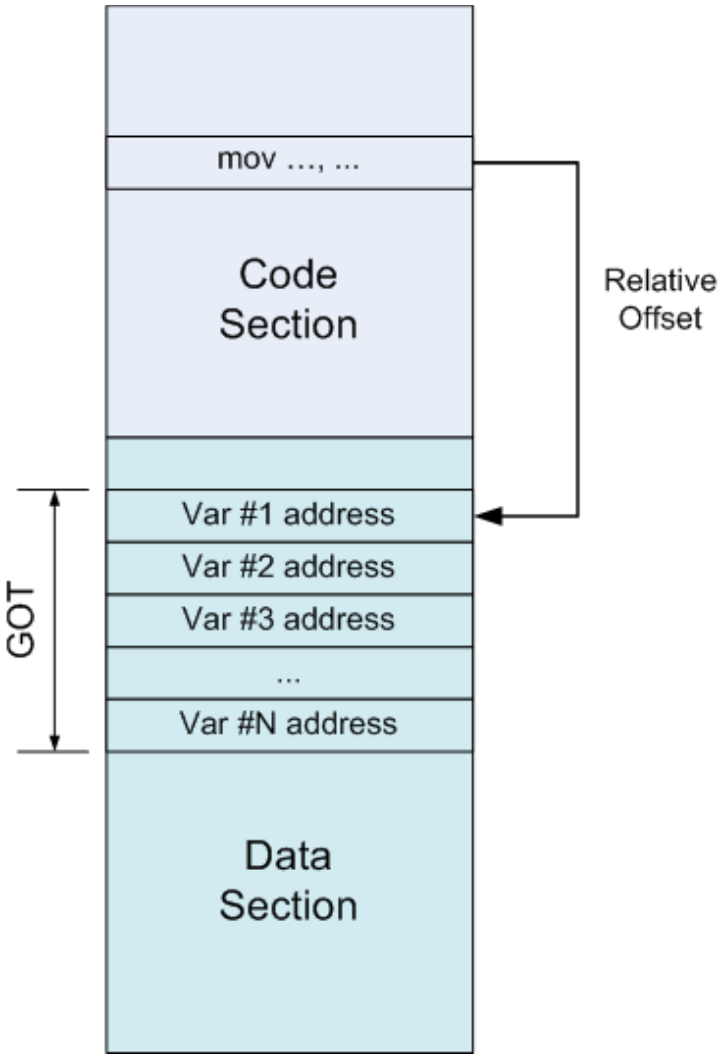
What happens here is:

1. The CPU executes `call TMPLABEL`, which causes it to save the address of the next instruction (the `pop ebx`) on stack and jump to the label.
2. Since the instruction at the label is `pop ebx`, it gets executed next. It pops a value from the stack into `ebx`. But this value is the address of the instruction itself, so `ebx` now effectively contains the value of the instruction pointer.

The Global Offset Table (GOT)

With this at hand, we can finally get to the implementation of position-independent data addressing on x86. It is accomplished by means of a "global offset table", or in short GOT.

A GOT is simply a table of addresses, residing in the data section. Suppose some instruction in the code section wants to refer to a variable. Instead of referring to it directly by absolute address (which would require a relocation), it refers to an entry in the GOT. Since the GOT is in a known place in the data section, this reference is relative and known to the linker. The GOT entry, in turn, will contain the absolute address of the variable:



In pseudo-assembly, we replace an absolute addressing instruction:

```
; Place the value of the variable in edx
mov edx, [ADDR_OF_VAR]
```

With displacement addressing from a register, along with an extra indirection:

```
; 1. Somehow get the address of the GOT into ebx
lea ebx, ADDR_OF_GOT

; 2. Suppose ADDR_OF_VAR is stored at offset 0x10
;    in the GOT. Then this will place ADDR_OF_VAR
;    into edx.
mov edx, DWORD PTR [ebx + 0x10]

; 3. Finally, access the variable and place its
;    value into edx.
mov edx, DWORD PTR [edx]
```

So, we've gotten rid of a relocation in the code section by redirecting variable references through the GOT. But we've also created a relocation in the data section. Why? Because the GOT still has to contain the absolute address of the variable for the scheme described above to work. So what have we gained?

A lot, it turns out. A relocation in the data section is much less problematic than one in the code section, for two reasons (which directly address the two main problems of load-time relocation of code described in the beginning of the article):

1. Relocations in the code section are required *per variable reference*, while in the GOT we only need to relocate once *per variable*. There are likely much more references to variables than variables, so this is more efficient.
2. The data section is writable and not shared between processes anyway, so adding relocations to it does no harm. Moving relocations from the code section, however, allows to make it read-only and share it between processes.

PIC with data references through GOT - an example

I will now show a complete example that demonstrates the mechanics of PIC:

```
int myglob = 42;

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

This chunk of code will be compiled into a shared library (using the `-fpic` and `-shared` flags as appropriate) named `libmlpic_dataonly.so`.

Let's take a look at its disassembly, focusing on the `ml_func` function:

```
0000043c <ml_func>:
 43c:   55                      push   ebp
 43d:   89 e5                   mov    ebp,esp
 43f:   e8 16 00 00 00          call   45a <__i686.get_pc_thunk.cx>
 444:   81 c1 b0 1b 00 00       add    ecx,0x1bb0
 44a:   8b 81 f0 ff ff ff       mov    eax,DWORD PTR [ecx-0x10]
 450:   8b 00                   mov    eax,DWORD PTR [eax]
 452:   03 45 08                add    eax,DWORD PTR [ebp+0x8]
 455:   03 45 0c                add    eax,DWORD PTR [ebp+0xc]
 458:   5d                      pop    ebp
 459:   c3                      ret

0000045a <__i686.get_pc_thunk.cx>:
 45a:   8b 0c 24                mov    ecx,DWORD PTR [esp]
 45d:   c3                      ret
```

I'm going to refer to instructions by their addresses (the left-most number in the disassembly). This address is the offset from the load address of the shared library.

- At `43f`, the address of the next instruction is placed into `ecx`, by means of the technique described in the "key insight #2" section above.
- At `444`, a known constant offset from the instruction to the place where the GOT is located is added to `ecx`. So `ecx` now serves as a base pointer to GOT.
- At `44a`, a value is taken from `[ecx - 0x10]`, which is a GOT entry, and placed into `eax`. This is the address of `myglob`.
- At `450` the indirection is done, and the *value* of `myglob` is placed into `eax`.
- Later the parameters `a` and `b` are added to `myglob` and the value is returned (by keeping it in `eax`).

We can also query the shared library with `readelf -S` to see where the GOT section was placed:

```
Section Headers:
  [Nr] Name     Type            Addr     Off    Size   ES Flg Lk Inf Al
  <snip>
  [19] .got     PROGBITS        00001fe4 000fe4 000010 04  WA  0   0  4
  [20] .got.plt PROGBITS        00001ff4 000ff4 000014 04  WA  0   0  4
  <snip>
```

Let's do some math to check the computation done by the compiler to find `myglob`. As I mentioned above, the call to `__i686.get_pc_thunk.cx` places the address of the next instruction into `ecx`. That address is `0x444` [2]. The next instruction then adds `0x1bb0` to it, and the result in `ecx` is going to be `0x1ff4`. Finally, to actually obtain the GOT entry holding the address of `myglob`, displacement addressing is used - `[ecx - 0x10]`, so the entry is at `0x1fe4`, which is the first entry in the GOT according to the section header.

Why there's another section whose name starts with `.got` will be explained later in the article [3]. Note that the compiler chooses to point `ecx` to after the GOT and then use negative offsets to obtain entries. This is fine, as long as the math works out. And so far it does.

There's something we're still missing, however. How does the address of `myglob` actually get into the GOT slot at `0x1fe4`? Recall that I mentioned a relocation, so let's find it:

```
> readelf -r libmlpic_dataonly.so

Relocation section '.rel.dyn' at offset 0x2dc contains 5 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00002008  00000008 R_386_RELATIVE
00001fe4  00000406 R_386_GLOB_DAT    0000200c   myglob
<snip>
```

Note the relocation section for `myglob`, pointing to address `0x1fe4`, as expected. The relocation is of type `R_386_GLOB_DAT`, which simply tells the dynamic loader - "put the actual value of the symbol (i.e. its address) into that offset". So everything works out nicely. All that's left is to check how it actually looks when the library

is loaded. We can do this by writing a simple "driver" executable that links to `libmlpic_dataonly.so` and calls `ml_func`, and then running it through GDB.

```
> gdb driver
[...] skipping output
(gdb) set environment LD_LIBRARY_PATH=.
(gdb) break ml_func
[...]
(gdb) run
Starting program: [...]pic_tests/driver

Breakpoint 1, ml_func (a=1, b=1) at ml_reloc_dataonly.c:5
5         return myglob + a + b;
(gdb) set disassembly-flavor intel
(gdb) disas ml_func
Dump of assembler code for function ml_func:
   0x0013143c <+0>:   push   ebp
   0x0013143d <+1>:   mov    ebp,esp
   0x0013143f <+3>:   call   0x13145a <__i686.get_pc_thunk.cx>
   0x00131444 <+8>:   add    ecx,0x1bb0
=> 0x0013144a <+14>:  mov    eax,DWORD PTR [ecx-0x10]
   0x00131450 <+20>:  mov    eax,DWORD PTR [eax]
   0x00131452 <+22>:  add    eax,DWORD PTR [ebp+0x8]
   0x00131455 <+25>:  add    eax,DWORD PTR [ebp+0xc]
   0x00131458 <+28>:  pop    ebp
   0x00131459 <+29>:  ret
End of assembler dump.
(gdb) i registers
eax            0x1    1
ecx            0x132ff4       1257460
[...] skipping output
```

The debugger has entered `ml_func`, and stopped at IP `0x0013144a` [4]. We see that `ecx` holds the value `0x132ff4` (which is the address of the instruction plus `0x1bb0`, as explained before). Note that at this point, at runtime, these are absolute addresses - the shared library has already been loaded into the address space of the process.

So, the GOT entry for `myglob` is at `[ecx - 0x10]`. Let's check what's there:

```
(gdb) x 0x132fe4
0x132fe4:     0x0013300c
```

So, we'd expect `0x0013300c` to be the address of `myglob`. Let's verify:

```
(gdb) p &myglob
$1 = (int *) 0x13300c
```

Indeed, it is!

Function calls in PIC

Alright, so this is how data addressing works in position independent code. But what about function calls? Theoretically, the exact same approach could work for function calls as well. Instead of `call` actually containing the address of the function to call, let it contain the address of a known GOT entry, and fill in that entry during loading.

But this is *not* how function calls work in PIC. What actually happens is a bit more complicated. Before I explain how it's done, a few words about the motivation for such a mechanism.

The lazy binding optimization

When a shared library refers to some function, the real address of that function is not known until load time. Resolving this address is called *binding*, and it's something the dynamic loader does when it loads the shared library into the process's memory space. This binding process is non-trivial, since the loader has to actually *look up* the function symbol in special tables [5].

So, resolving each function takes time. Not a lot of time, but it adds up since the amount of functions in libraries is typically much larger than the amount of global variables. Moreover, most of these resolutions are done in vain, because in a typical run of a program only a fraction of functions actually get called (think about various functions handling error and special conditions, which typically don't get called at all).

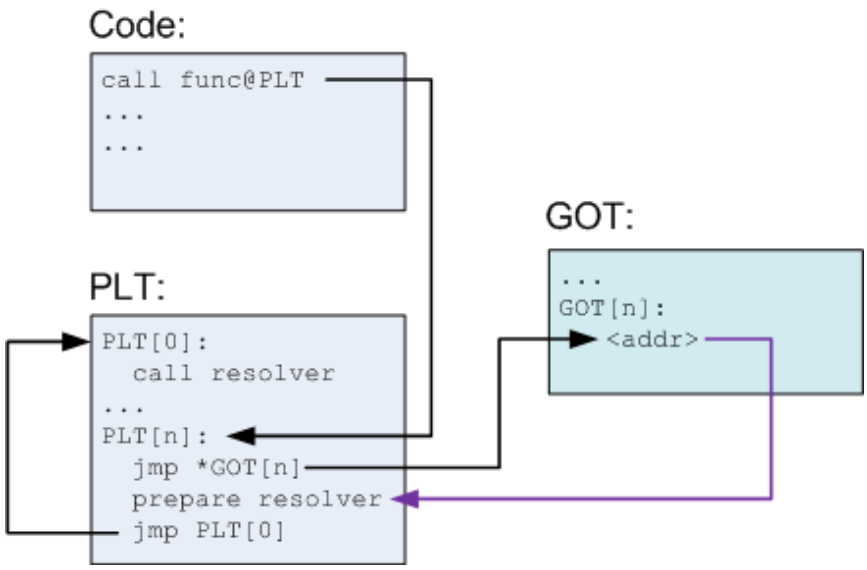
So, to speed up this process, a clever lazy binding scheme was devised. "Lazy" is a generic name for a family of optimizations in computer programming, where work is delayed until the last moment when it's actually needed, with the intention of avoiding doing this work if its results are never required during a specific run of a program. Good examples of laziness are copy-on-write (http://en.wikipedia.org/wiki/Copy-on-write) and lazy evaluation (http://en.wikipedia.org/wiki/Lazy_evaluation).

This lazy binding scheme is attained by adding yet another level of indirection - the PLT.

The Procedure Linkage Table (PLT)

The PLT is part of the executable text section, consisting of a set of entries (one for each external function the shared library calls). Each PLT entry is a short chunk of executable code. Instead of calling the function directly, the code calls an entry in the PLT, which then takes care to call the actual function. This arrangement is sometimes called a "trampoline (http://en.wikipedia.org/wiki/Trampoline_%28computing%29)". Each PLT entry also has a corresponding entry in the GOT which contains the actual offset to the function, but only when the dynamic loader resolves it. I know this is confusing, but hopefully it will be come clearer once I explain the details in the next few paragraphs and diagrams.

As the previous section mentioned, PLTs allow lazy resolution of functions. When the shared library is first loaded, the function calls have not been resolved yet:



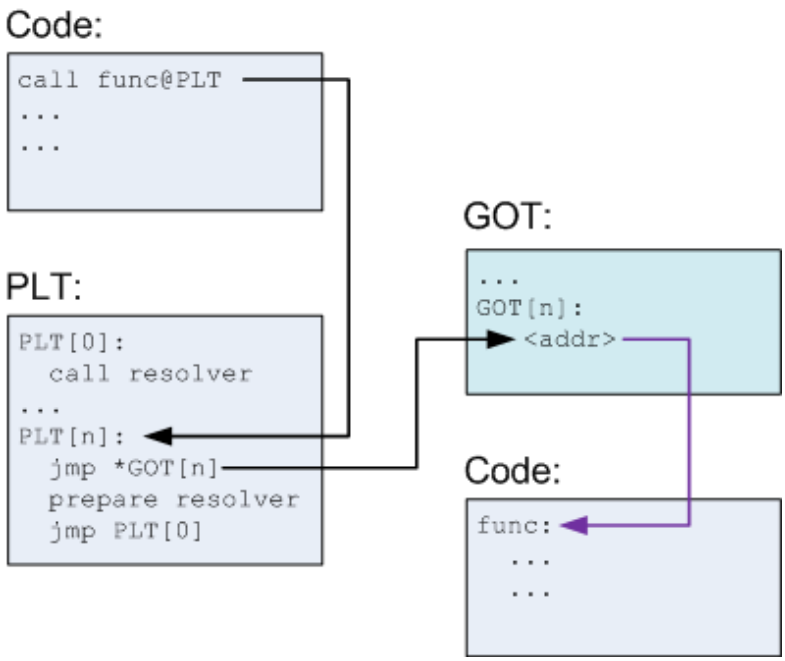
Explanation:

- In the code, a function `func` is called. The compiler translates it to a call to `func@plt`, which is some N-th entry in the PLT.
- The PLT consists of a special first entry, followed by a bunch of identically structured entries, one for each function needing resolution.
- Each PLT entry but the first consists of these parts:
  - A jump to a location which is specified in a corresponding GOT entry
  - Preparation of arguments for a "resolver" routine
  - Call to the resolver routine, which resides in the first entry of the PLT
- The first PLT entry is a call to a resolver routine, which is located in the dynamic loader itself [6]. This routine resolves the actual address of the function. More on its action a bit later.
- Before the function's actual address has been resolved, the Nth GOT entry just points to after the jump. This is why this arrow in the diagram is colored differently - it's not an actual jump, just a pointer.

What happens when `func` is called for the first time is this:

- `PLT[n]` is called and jumps to the address pointed to in `GOT[n]`.
- This address points into `PLT[n]` itself, to the preparation of arguments for the resolver.
- The resolver is then called.
- The resolver performs resolution of the actual address of `func`, places its actual address into `GOT[n]` and calls `func`.

After the first call, the diagram looks a bit differently:



Note that `GOT[n]` now points to the actual `func` [7] instead of back into the PLT. So, when `func` is called again:

- `PLT[n]` is called and jumps to the address pointed to in `GOT[n]`.
- `GOT[n]` points to `func`, so this just transfers control to `func`.

In other words, now `func` is being actually called, without going through the resolver, at the cost of one additional jump. That's all there is to it, really. This mechanism allows lazy resolution of functions, and no resolution at all for functions that aren't actually called.

It also leaves the code/text section of the library completely position independent, since the only place where an absolute address is used is the GOT, which resides in the data section and will be relocated by the dynamic loader. Even the PLT itself is PIC, so it can live in the read-only text section.

I didn't get into much details regarding the resolver, but it's really not important for our purpose here. The resolver is simply a chunk of low-level code in the loader that does symbol resolution. The arguments prepared for it in each PLT entry, along with a suitable relocation entry, help it know about the symbol that needs resolution and about the GOT entry to update.

PIC with function calls through PLT and GOT - an example

Once again, to fortify the hard-learned theory with a practical demonstration, here's a complete example showing function call resolution using the mechanism described above. I'll be moving forward a bit faster this time.

Here's the code for the shared library:

```
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;
}
```

This code will be compiled into `libmlpic.so`, and the focus is going to be on the call to `ml_util_func` from `ml_func`. Let's first disassemble `ml_func`:

```
00000477 <ml_func>:
 477:   55                      push   ebp
 478:   89 e5                   mov    ebp,esp
 47a:   53                      push   ebx
 47b:   83 ec 24                sub    esp,0x24
 47e:   e8 e4 ff ff ff          call   467 <__i686.get_pc_thunk.bx>
 483:   81 c3 71 1b 00 00       add    ebx,0x1b71
 489:   8b 45 08                mov    eax,DWORD PTR [ebp+0x8]
 48c:   89 04 24                mov    DWORD PTR [esp],eax
 48f:   e8 0c ff ff ff          call   3a0 <ml_util_func@plt>
 <... snip more code>
```

The interesting part is the call to `ml_util_func@plt`. Note also that the address of GOT is in `ebx`. Here's what `ml_util_func@plt` looks like (it's in an executable section called `.plt`):

```
000003a0 <ml_util_func@plt>:
 3a0:   ff a3 14 00 00 00       jmp    DWORD PTR [ebx+0x14]
 3a6:   68 10 00 00 00          push   0x10
 3ab:   e9 c0 ff ff ff          jmp    370 <_init+0x30>
```

Recall that each PLT entry consists of three parts:

- A jump to an address specified in GOT (this is the jump to `[ebx+0x14]`)
- Preparation of arguments for the resolver
- Call to the resolver

The resolver (PLT entry 0) resides at address `0x370`, but it's of no interest to us here. What's more interesting is to see what the GOT contains. For that, we first have to do some math.

The "get IP" trick in `ml_func` was done on address `0x483`, to which `0x1b71` is added. So the base of the GOT is at `0x1ff4`. We can take a peek at the GOT contents with `readelf` [8]:

```
> readelf -x .got.plt libmlpic.so

Hex dump of section '.got.plt':
  0x00001ff4 241f0000 00000000 00000000 86030000 $...............
  0x00002004 96030000 a6030000                    ........
```

The GOT entry `ml_util_func@plt` looks at is at offset `+0x14`, or `0x2008`. From above, the word at that location is `0x3a6`, which is the address of the `push` instruction in `ml_util_func@plt`.

To help the dynamic loader do its job, a relocation entry is also added and specifies which place in the GOT to relocate for `ml_util_func`:

```
> readelf -r libmlpic.so
[...] snip output

Relocation section '.rel.plt' at offset 0x328 contains 3 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00002000  00000107 R_386_JUMP_SLOT   00000000   __cxa_finalize
00002004  00000207 R_386_JUMP_SLOT   00000000   __gmon_start__
00002008  00000707 R_386_JUMP_SLOT   0000046c   ml_util_func
```

The last line means that the dynamic loader should place the value (address) of symbol `ml_util_func` into `0x2008` (which, recall, is the GOT entry for this function).

It would be interesting to see this GOT entry modification actually happen after the first call. Let's once again use GDB for the inspection.

```
> gdb driver
[...] skipping output
(gdb) set environment LD_LIBRARY_PATH=.
(gdb) break ml_func
Breakpoint 1 at 0x80483c0
(gdb) run
Starting program: /pic_tests/driver

Breakpoint 1, ml_func (a=1, b=1) at ml_main.c:10
10        int c = b + ml_util_func(a);
(gdb)
```

We're now before the first call to `ml_util_func`. Recall that GOT is pointed to by `ebx` in this code. Let's see what's in it:

```
(gdb) i registers ebx
ebx            0x132ff4
```

And the offset to the entry we need is at `[ebx+0x14]`:

```
(gdb) x/w 0x133008
0x133008:     0x001313a6
```

Yep, the `0x3a6` ending, looks right. Now, let's step until after the call to `ml_util_func` and check again:

```
(gdb) step
ml_util_func (a=1) at ml_main.c:5
5         return a + 1;
(gdb) x/w 0x133008
0x133008:     0x0013146c
```

The value at `0x133008` was changed. Hence, `0x0013146c` should be the real address of `ml_util_func`, placed in there by the dynamic loader:

```
(gdb) p &ml_util_func
$1 = (int (*)(int)) 0x13146c <ml_util_func>
```

Just as expected.

Controlling if and when the resolution is done by the loader

This would be a good place to mention that the process of lazy symbol resolution performed by the dynamic loader can be configured with some environment variables (and corresponding flags to `ld` when linking the shared library). This is sometimes useful for special performance requirements or debugging.

The `LD_BIND_NOW` env var, when defined, tells the dynamic loader to always perform the resolution for all symbols at start-up time, and not lazily. You can easily verify this in action by setting this env var and re-running the previous sample with GDB. You'll see that the GOT entry for `ml_util_func` contains its real address even before the first call to the function.

Conversely, the `LD_BIND_NOT` env var tells the dynamic loader not to update the GOT entry at all. Each call to an external function will then go through the dynamic loader and be resolved anew.

The dynamic loader is configurable by other flags as well. I encourage you to go over `man ld.so` - it contains some interesting information.

The costs of PIC

This article started by stating the problems of load-time relocation and how the PIC approach fixes them. But PIC is also not without problems. One immediately apparent cost is the extra indirection required for all external references to data and code in PIC. That's an extra memory load for each reference to a global variable, and for each call to a function. How problematic this is in practice depends on the compiler, the CPU architecture and the particular application.

Another, less apparent cost, is the increased register usage required to implement PIC. In order to avoid locating the GOT too frequently, it makes sense for the compiler to generate code that keeps its address in a register (usually `ebx`). But that ties down a whole register just for the sake of GOT. While not a big problem for RISC architectures that tend to have a lot of general purposes registers, it presents a performance problem for architectures like x86, which has a small amount of registers. PIC means having one general purpose register less, which adds up indirect costs since now more memory references have to be made.

Conclusion

This article explained what position independent code is, and how it helps create shared libraries with shareable read-only text sections. There are some tradeoffs when choosing between PIC and its alternative (load-time relocation), and the eventual outcome really depends on a lot of factors, like the CPU architecture on which the program is going to run.

That said, PIC is becoming more and more popular. Some non-Intel architectures like SPARC64 force PIC-only code for shared libraries, and many others (for example, ARM) include IP-relative addressing modes to make PIC more efficient. Both are true for the successor of x86, the x64 architecture. I will discuss PIC on x64 in a future article.

The focus of this article, however, has not been on performance considerations or architectural decisions. My aim was to explain, given that PIC is used, *how it works*. If the explanation wasn't clear enough - please let me know in the comments and I will try to provide more information.

---

[1] Unless all applications load this library into the exact same virtual memory address. But this usually isn't done on Linux.

[2] `0x444` (and all other addresses mentioned in this computation) is relative to the load address of the shared library, which is unknown until an executable actually loads it at runtime. Note how it doesn't matter in the code since it only juggles *relative* addresses.

[3] The astute reader may wonder why `.got` is a separate section at all. Didn't I just show in the diagrams that it's located in the data section? In practice, it is. I don't want to get into the distinction between ELF sections and segments here, since that would take use too far away from the point. But briefly, any number of "data" sections can be defined for a library and mapped into a read-write segment. This doesn't really matter, as long as the ELF file is organized correctly. Separating the data segment into different logical sections provides modularity and makes the linker's job easier.

[4] Note that `gdb` skipped the part where `ecx` is assigned. That's because it's kind-of considered to be part of the function's prolog (the real reason is in the way `gcc` structures its debug info, of course). Several references to global data and functions are made inside a function, and a register pointing to GOT can serve all of them.

[5] Shared library ELF objects actually come with special hash table sections for this purpose.

[6] The dynamic loader on Linux is just another shared library which gets loaded into the address space of all running processes.

[7] I placed `func` in a separate code section, although in theory this could be the same one where the call to `func` is made (i.e. in the same shared library). The "extra credit" section of this article (https://eli.thegreenplace.net/2011/08/25/load-time-relocation-of-shared-libraries/) has information about why a call to an external function in the same shared library needs PIC (or relocation) as well.

[8] Recall that in the data reference example I promised to explain why there are apparently two GOT sections in the object: `.got` and `.got.plt`. Now it should become obvious that this is just to conveniently split the GOT entries required for global data from GOT entries required for the PLT. This is also why when the GOT offset is computed in functions, it points to `.got.plt`, which comes right after `.got`. This way, negative offsets lead us to `.got`, while positive offsets lead us to `.got.plt`. While convenient, such an arrangement is by no means compulsory. Both parts could be placed into a single `.got` section.

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

↑ Back to top