How is glibc loaded at runtime?

I've been looking into address-space layout randomization (ASLR). ASLR relies on randomizing the base address of things like shared libraries, making return-to-libc attacks more difficult.

I understood the basics of ASLR but I still had a lot of questions. How are shared libraries, like libc, loaded at runtime? What is the global offset table? What is the procedure linkage table? What is a position independent executable? In this post, we're going to look at all of these.

Back in the Day

In "the olden days" libraries used to be hard coded to be loaded at a fixed address in memory space. Runtime linkers had to deal with relocating conflicting hard coded addresses. Windows, to some extent, still does this.

PIC - Position Independent Code

Then came along Position Independent Code which simply means that the code (usually shared libraries) can be loaded at any address in memory-space and relocations are no longer a problem. In order to do that, binaries added sections for the GOT and the PLT.

Global Offset Table

Every ELF executable has a section called the Global Offset Table or the GOT for short. This table is responsible for holding the *absolute address* of functions in shared libraries linked dynamically at runtime.

```
nobody@nobody:~$ objdump -R ./hello_world

./hello_world:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049564 R_386_GLOB_DAT    __gmon_start__
```

```
08049574 R_386_JUMP_SLOT   __gmon_start__
08049578 R_386_JUMP_SLOT   __libc_start_main
0804957c R_386_JUMP_SLOT   printf
```

Procedure Linkage Table

Just like the GOT, every ELF executable also has a section called the Procedure Linkage Table or PLT for short (not to be confused with BLT (Bacon Lettuce Tomato) :-)). If you've read disassembled code, you'll often see function calls like `printf@plt`. That's a call to the `printf` in the procedure linking table. The PLT is sort of like the spring board that allows us to resolve the absolute addresses of shared libraries at runtime.

```
nobody@nobody:~$ objdump -d -j .plt ./hello_world

./hello_world:     file format elf32-i386

Disassembly of section .plt:

08048270 <__gmon_start__@plt-0x10>:
 8048270:       ff 35 6c 95 04 08       pushl  0x804956c
 8048276:       ff 25 70 95 04 08       jmp    *0x8049570
 804827c:       00 00                   add    %al,(%eax)

08048280 <__gmon_start__@plt>:
 8048280:       ff 25 74 95 04 08       jmp    *0x8049574
 8048286:       68 00 00 00 00          push   $0x0
 804828b:       e9 e0 ff ff ff          jmp    8048270 <_init+0x18>

08048290 <__libc_start_main@plt>:
 8048290:       ff 25 78 95 04 08       jmp    *0x8049578
 8048296:       68 08 00 00 00          push   $0x8
 804829b:       e9 d0 ff ff ff          jmp    8048270 <_init+0x18>
```

```
080482a0 <printf@plt>:
 80482a0:       ff 25 7c 95 04 08       jmp    *0x804957c
 80482a6:       68 10 00 00 00          push   $0x10
 80482ab:       e9 c0 ff ff ff          jmp    8048270 <_init+0x18>
```

The GOT, The PLT, and the Linker

How do these all work together to load a shared library at runtime? Well it's actually pretty cool. Lets walk through the *first* call to `printf`. `printf@plt`, which is not really `printf` but a location in the PLT, is called and the first jump is executed.


```
080482a0 <printf@plt>:
 80482a0:       ff 25 7c 95 04 08       jmp    *0x804957c
 80482a6:       68 10 00 00 00          push   $0x10
 80482ab:       e9 c0 ff ff ff          jmp    8048270 <_init+0x18>
```


Notice that this jump is a pointer to an address. We're going to jump to the address pointed to by this address. The `0x804957c` is an address in the GOT. The GOT will eventually hold the absolute address call to `printf`, *however*, on the very first call the address will point back to the instruction after the jump in the PLT - `0x80482a6`. We can see this below by looking at the output of the GOT. Essentially we'll execute all of the instructions of the `printf@plt` the very first call.

```
(gdb) x/8x 0x804957c-20
0x8049568 <_GLOBAL_OFFSET_TABLE_>:      0x0804949c      0xb80016e0
0xb7ff92f0      0x08048286
```

```
0x8049578 <_GLOBAL_OFFSET_TABLE_+16>:    0xb7eafde0      0x080482a6
0x00000000      0x00000000
```

In the PLT code, an offset is pushed onto the stack and another `jmp` is executed

```
080482a0 <printf@plt>:
 80482a0:       ff 25 7c 95 04 08       jmp    *0x804957c
 80482a6:       68 10 00 00 00          push   $0x10
 80482ab:       e9 c0 ff ff ff          jmp    8048270 <_init+0x18>
```

This jump is a jump into the eventual runtime linker code that will load the shared library which contains printf. The offset, `$0x10`, that was pushed onto the stack tells the linker code the offset of the symbol in the relocation table (see `objdump -R ./hello_world` output above), `printf` in this case. The linker will then write the address of `printf` into the GOT at `0x804957c`. We can see this if we look at the GOT after the library has been loaded.

```
(gdb) x/8x 0x804957c-20
0x8049568 <_GLOBAL_OFFSET_TABLE_>:       0x0804949c      0xb80016e0
0xb7ff92f0      0x08048286
0x8049578 <_GLOBAL_OFFSET_TABLE_+16>:    0xb7eafde0      0xb7edf620
0x00000000      0x00000000
```

Notice that the previous address, `0x80482a6`, has been replaced by the linker with 0xb7edf620. To confirm that this indeed is the address for printf, we can start a disassemble at this address

```
(gdb) disassemble 0xb7edf620
Dump of assembler code for function printf:
...
```

Since the library is now loaded and the GOT has been overwritten with the absolute address to printf, subsequent calls to the function `printf@plt` will jump directly to the address of printf! All of this also has the added benefit that a shared library is not loaded until a function in it's library is loaded – in other words, a nice form of "lazy-loading!"