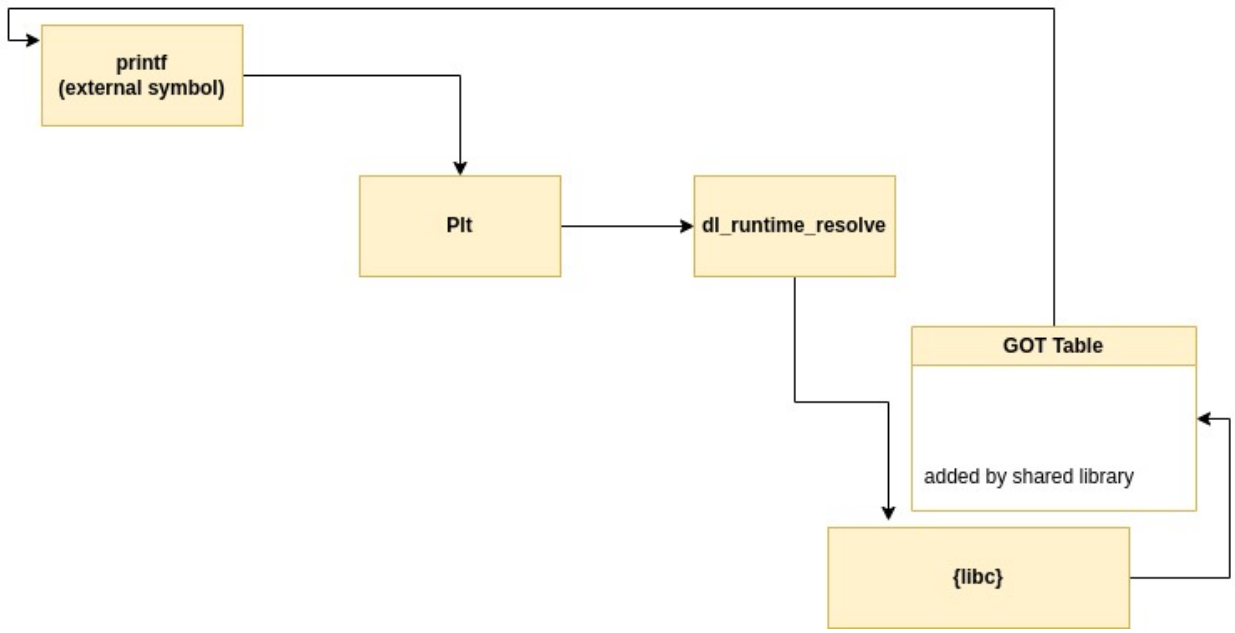
CSE Weekly Digest

Got and Plt section internal

How Share library Work



by Tiwari Abhinav Ashok Kumar on November 06

This document is all about the got and plt section internally. Hope it will help you in understanding it.

Example shown below will be on x86 Linux platform but all the logic and concept can be applied on all the architecture in similar or different ways. (These ideas are based on ELF linking and glibc).

High-Level Introduction

When you first hear about **got** and **plt** your mind is not easily able to digest this concept.

Let's Discuss What all this nonsense is?

We all are aware that there are two type of binaries 👍

1. **Static linked binary** : Which is a self contained binary. Which means it contains all the information which is required by an executable to get executed without depending on other external libraries.
2. **Dynamic Linked binaries :** These are default generated binaries by any of the compilers. These binaries dont consist of a lot of function definitions and they depend on the system libraries which provide definition of the function. (e.g printf depends on the system C library).

In the dynamic linked library our program needs to get the address of printf to call it.

In order to locate these functions, your program needs to know the address of printf to call it. But these strategies have some problems.

Each time when we change the library the address of the function inside the library will change. You have to now rebuild every binary on your system which will be a very hectic task.

To solve these upcoming system or modern system usage, ASLR loads libraries at a different location on each program invocation.

When hard code addresses this possibility will be rendered.

This allows a developer to develop a strategy to allow looking up all the addresses when the program was run and giving an algorithm or mechanism to call these functions from libraries. This strategy is known as relocation.

We are also aware that relocation is performed by the linker during runtime. Each and every program which builds dynamically they will get linked against the linker. This gets set in a different area in memory layout of the ELF known as .interp.

Relocations

When you see the ELF file you will find there are various sections inside it and relocation requires many of these sections.

Let's first define the section and then we will discuss how they are being used.

.got

Its abbreviation is Global Offset Table. This table actually consist of offsets for the external symbol.(The way linker have filled it)

This is the GOT, or Global Offset Table. This is the actual table of offsets as filled in by the linker for external symbols.

.plt

Its abbreviation is Procedure Linkage Table . These are nothing but a small portion of code that looks up at the addresses which reside inside the .got.plt section after looking up the address, if the address is correct then it jumps to that address , or it triggers the code in the linker to look up the address. (If the address has not been filled in to .got.plt yet.). This small portion of the code is known as stubs.

.got.plt

This is the GOT for the PLT. This area contains the target addresses (after they have looked up) .or an address back in the .plt to trigger the lookup. Classically, this data was part of the .got section.

.plt.got

This section contains all the permutations and combinations of PLT and GOT sections. This section contains a code to jump to the first location of the .got .

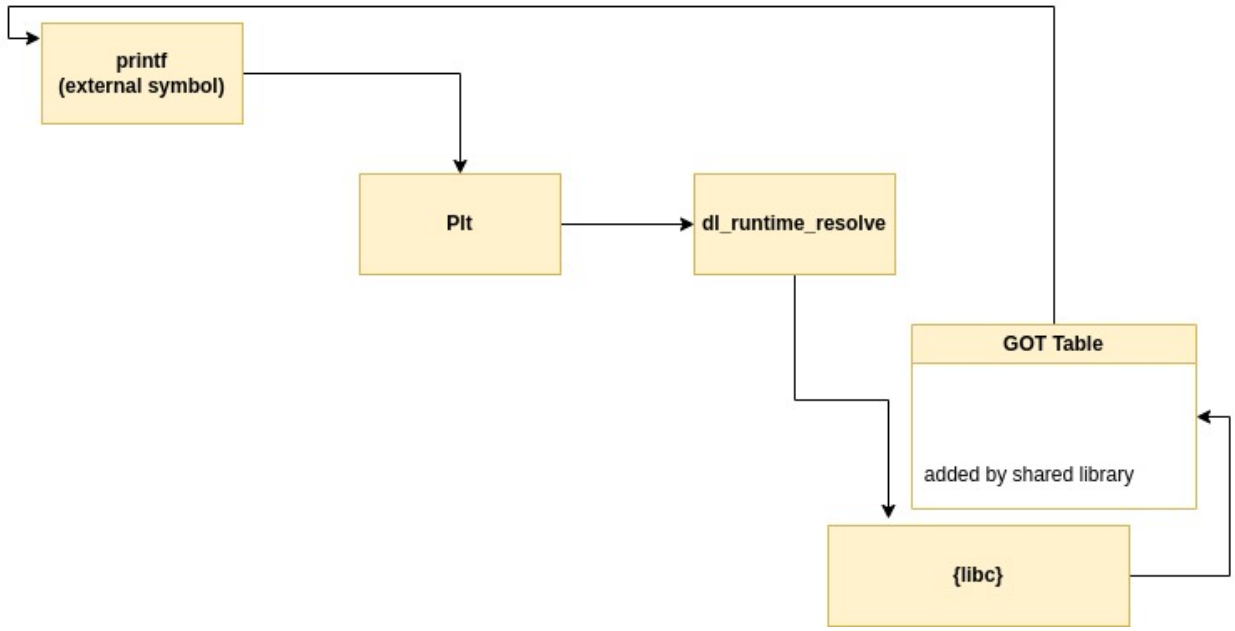
TL;DR: Those starting with .plt contain stubs to jump to the target, those starting with .got are tables of the target addresses.

Let's look at an example of how these sections interact with each other. We will use a simple C code which will call printf and exiit as an external function.

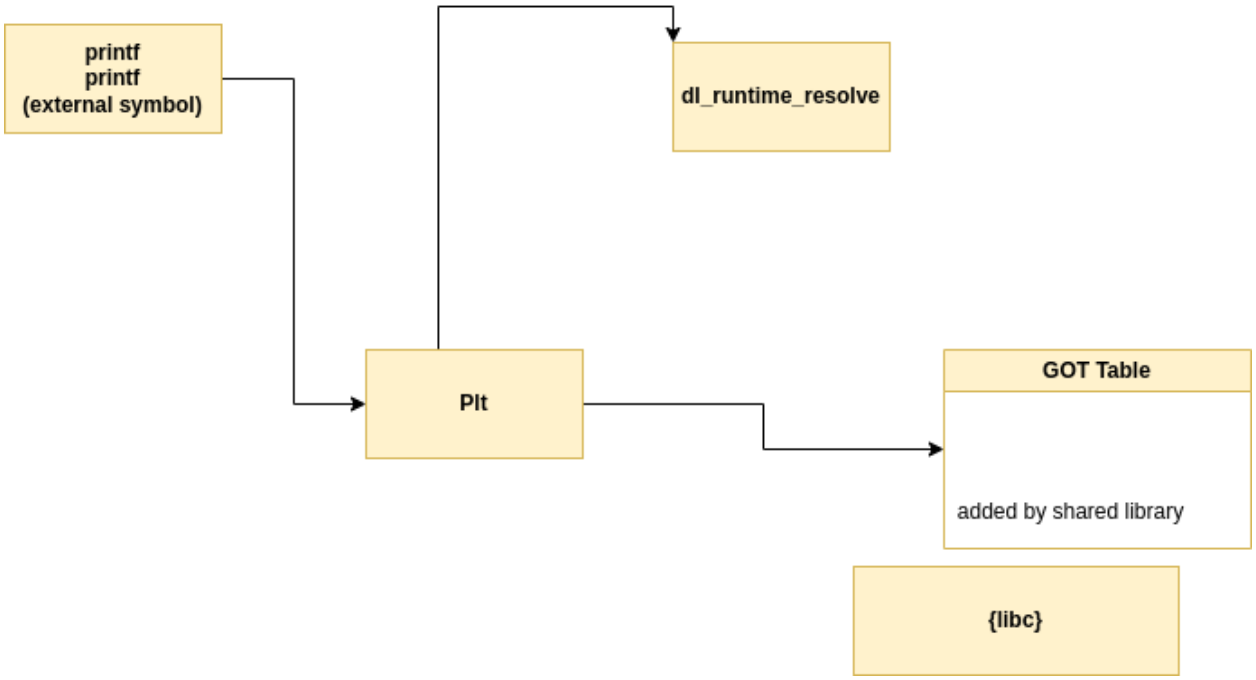
When we call functions like printf they are dynamically link to the binary and then executed.

When we call a function from the shared library like libc a call to piece of code **PLT (Procedural linkage table)** is made.

This PLT will take help of the **dl_runtime_resolve linker** which will search through libc library for the printf function and it will add an entry inside the Global Offset table (GOT) with the address of function invoked once the address is resolved it will get executed and control will transfer to the saved return address.



When the same function is called for the next time in the code the address stored in GOT section will be used instead of invoking dynamic linker to resolve the address again .



To summarize plt is just a stub code to invoke this dynamic linker(dl_runtime_resolve) to resolve the address of function that are being invoked by libc .Once address is resolved its going to get an entry in Global Offset Table . If the same function is invoked for the next time instead of resolving the address again address will be taken from the global offset table.

```
//Build With: gcc -m32 -no-pie -g -o plt plt.c
```

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
        printf("Hello World! \n");
        exit(0);
}
```

Before moving further lets examine section headers

Execute :- **readelf -S plt**

There are 35 section headers, starting at offset 0x3834:

Section Headers:

[Nr] Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[12] .plt	PROGBITS	08049030	001030	000040	04	AX	0	0	16
[13] .text	PROGBITS	08049070	001070	00014c	00	AX	0	0	16
[21] .got	PROGBITS	0804bffc	002ffc	000004	04	WA	0	0	4
[22] .got.plt	PROGBITS	0804c000	003000	000018	04	WA	0	0	4
[9] .rel.dyn	REL	08048304	000304	000008	08	A	5	0	4
[10] .rel.plt	REL	0804830c	00030c	000018	08	AI	5	22	4

This only consists of the section which I am going to discuss

Note:- ***"So let's walk through this process with the use of GDB. (I'm using the fantastic GDB environment provided by pwndbg, so some UI elements might look a bit different from vanilla GDB.) We'll load up our binary and set a breakpoint just before puts gets called and then examine the flow step-by-step:"***

$ gdb plt

```
pwndbg> disass main
Dump of assembler code for function main:
   0x08049186 <+0>:     lea    ecx,[esp+0x4]
   0x0804918a <+4>:     and    esp,0xfffffff0
   0x0804918d <+7>:     push   DWORD PTR [ecx-0x4]
   0x08049190 <+10>:    push   ebp
   0x08049191 <+11>:    mov    ebp,esp
   0x08049193 <+13>:    push   ebx
   0x08049194 <+14>:    push   ecx
   0x08049195 <+15>:    call   0x80490c0 <__x86.get_pc_thunk.bx>
   0x0804919a <+20>:    add    ebx,0x2e66
   0x080491a0 <+26>:    sub    esp,0xc
   0x080491a3 <+29>:    lea    eax,[ebx-0x1ff8]
   0x080491a9 <+35>:    push   eax
   0x080491aa <+36>:    call   0x8049050 <puts@plt>
   0x080491af <+41>:    add    esp,0x10
   0x080491b2 <+44>:    sub    esp,0xc
   0x080491b5 <+47>:    push   0x0
   0x080491b7 <+49>:    call   0x8049060 <exit@plt>
End of assembler dump.
pwndbg>
```

We are calling puts let's place breakpoint their and check it

```
pwndbg> break *0x080491aa  #(Adress from which puts is called first time)
pwndbg> run
pwndbg> si            #(step to function until and unless you get actual  puts function)
pwndbg>x/i $pc   #(to check address where program counter is pointing)

=> 0x8049050 <puts@plt>:  jmp    DWORD PTR ds:0x804c010   #(puts function)
```

Here we can see that we are inside the plt section (**puts@plt**) and instruction is showing that it is performing **jmp (**jump**)** to a function pointer. The processor will dereference the pointer, then jump to the resulting address.

Let's check the difference and follow the jmp. Note that the pointer is in the **.got.plt** section as we described above.

```
pwndbg>x/xw 0x804c010
0x804c010 <puts@got.plt>:  0x08049056
pwndbg> x/i $pc
=> 0x8049050 <puts@plt>:  jmp    DWORD PTR ds:0x804c010
pwndbg> x/2i $pc
=> 0x8049056 <puts@plt+6>:         push   0x8
   0x804905b <puts@plt+11>:        jmp    0x8049030
```

Well, that's weird. We've just jumped to the next instruction! Why has this occurred? Well, it turns out that because we haven't called puts before, we need to trigger the first lookup. It pushes the slot number (0x0) on the stack, then calls the routine to lookup the symbol name. This happens to be the beginning of the .plt section. What does this stub do? Let's find out.

```
pwndbg>si
pwndbg> si
pwndbg> x/2i $pc
=> 0x8049030 <puts@plt>:  jmp    DWORD PTR ds:0x804c004
```

```
=> 0x8049036 <puts@plt>:  jmp    DWORD PTR ds:0x804c008
```

Now, we push the value of the second entry in .got.plt, then jump to the address stored in the third entry. Let's examine those values and carry on.

```
pwndbg>x/2wx 0x804c004
0x804c004:    0xf7ffda40    0xf7fd8fe0
```

Wait, where is that pointing? It turns out the first one points into the data segment of ld.so, and the 2nd into the executable area:

```
pwndbg>info sharedlibrary
From        To          Syms Read   Shared Object Library
0xf7fc7090  0xf7feb2a5  Yes         /lib/ld-linux.so.2
0xf7d98290  0xf7f18499  Yes         /lib/i386-linux-gnu/libc.so.6
```

Ah, finally, we're asking for the information for the puts symbol! These two addresses in the .got.plt section are populated by the linker/loader (ld.so) at the time it is loading the binary.

So, I'm going to treat what happens in ld.so as a black box. I encourage you to look into it, but exactly how it looks up the symbols is a little bit too low level for this post. Suffice it to say that eventually we will reach a ret from the ld.so code that resolves the symbol.

```
pwndbg> x/i $pc
=> 0xf7fd8fe0 <_dl_runtime_resolve>:    endbr32
pwndbg> info symbol $pc
_dl_runtime_resolve in section .text of /lib/ld-linux.so.2
```

Look at that, we find ourselves at puts, exactly where we'd like to be. Let's see how our stack looks at this point:

Put a break point at put as b put

```
pwndbg> info symbol $pc
=> 0xf7fdeb260 <_GI__IO_puts>:          endbr32
```

```
pwndbg> x/4w $esp
=> 0xffffd10c: 0x080491af    0x0804a008  0xf7fbe66c    0xf7fbeb10
pwndbg>x/s *(int *)($esp+4)
0x804a008:    "Hello World"
```

Absolutely no trace of the trip through .plt, ld.so, or anything but what you'd expect from a direct call to puts.

Share it and subscribe it