Linux Debugging（七）：使用反彙編理解動態庫函數調用方式GOT/PLT

By anzhsoft | Published 2014年3月6日

本文主要講解動態庫函數的地址是如何在運行時被定位的。首先介紹一下PIC和Relocatable的動態庫的區別。然後講解一下GOT和PLT的理論知識。GOT是Global Offset Table，是保存庫函數地址的區域。程序運行時，庫函數的地址會設置到GOT中。由於動態庫的函數是在使用時才被加載，因此剛開始GOT表是空的。地址的設置就涉及到了PLT，Procedure Linkage Table，它包含了一些代碼以調用庫函數，它可以被理解成一系列的小函數，這些小函數的數量其實就是庫函數的被使用到的函數的數量。簡單來說，PLT就是跳轉到GOT中所設置的地址而已。如果這個地址是空，那麼PLT的跳轉會巧妙的調用_dl_runtime_resolve去獲取最終地址並設置到GOT中去。由於庫函數的地址在運行時不會變，因此GOT一旦設置以後PLT就可以直接跳轉到庫函數的真實地址了。最後使用反彙編驗證和跳轉流程圖對上述結論加深理解。

1. 背景-PIC VS Relocatable

在Linux 下製作動態鏈接庫，“標準”的做法是編譯成位置無關代碼（Position Independent Code，PIC），然後鏈接成一個動態鏈接庫。那麼什麼是PIC呢？如果是非PIC的，那麼會有什麼問題？

(1) 可重定位代碼（relocatable code）：Windows DLL 以及不使用-fPIC 的Linux so。

生成動態庫時假定它被加載在地址0　處。加載時它會被加載到一個地址（base），這時要進行一次重定位（relocation），把代碼、數據段中所有的地址加上這個base　的值。這樣代碼運行時就能使用正確的地址了。當要再加載時根據加載到的位置再次重定位的。(因為它裡面的代碼並不是位置無關代碼)。因為so被每個程序加載的位置都不同,顯然這些重定位後的代碼也不同,當然不能共享。如果被多個應用程序共同使用,那麼它們必須每個程序維護一份so的代碼副本了。當然，主流現代操作系統都啟用了分頁內存機制，這使得重定位時可以使用

COW（copy on write）來節省內存（32 位Windows 就是這樣做的）；然而，頁面的粒度還是比較大的（例如IA32 上是4KiB），至少對於代碼段來說能節省的相當有限。不能共享就失去了共享庫的好處,實際上和靜態庫的區別並不大,在運行時佔用的內存是類似的,僅僅是二進制代碼佔的硬盤空間小一些。

(2) 位置無關代碼（position independent code）：使用-fPIC 的Linux so。

這樣的代碼本身就能被放到線性地址空間的任意位置，無需修改就能正確執行。通常的方法是獲取指令指針（如x86 的EIP 寄存器）的值，加上一個偏移得到全局變量/函數的地址。AMD64 下，必須使用位置無關代碼。x86下，在創建so時會有一個警告。但是這樣的so可以完全正常工作。PIC 的缺點主要就是代碼有可能長一些。例如x86，由於不能直接使用 [EIP+constant] 這樣的尋址方式，甚至不能直接將EIP 的值交給其他寄存器，要用到GOT（global

offset table）來定位全局變量和函數。這樣導致代碼的效率略低。PIC 的加載速度稍快，因為不需要做重定位。多個進程引用同一個PIC 動態庫時，可以共用內存。這一個庫在不同進程中的虛擬地址不同，但操作系統顯然會把它們映射到同一塊物理內存上。

因此，除非你的so不會被共享，否則還是加上-fPIC吧。

2. GOT和PLT

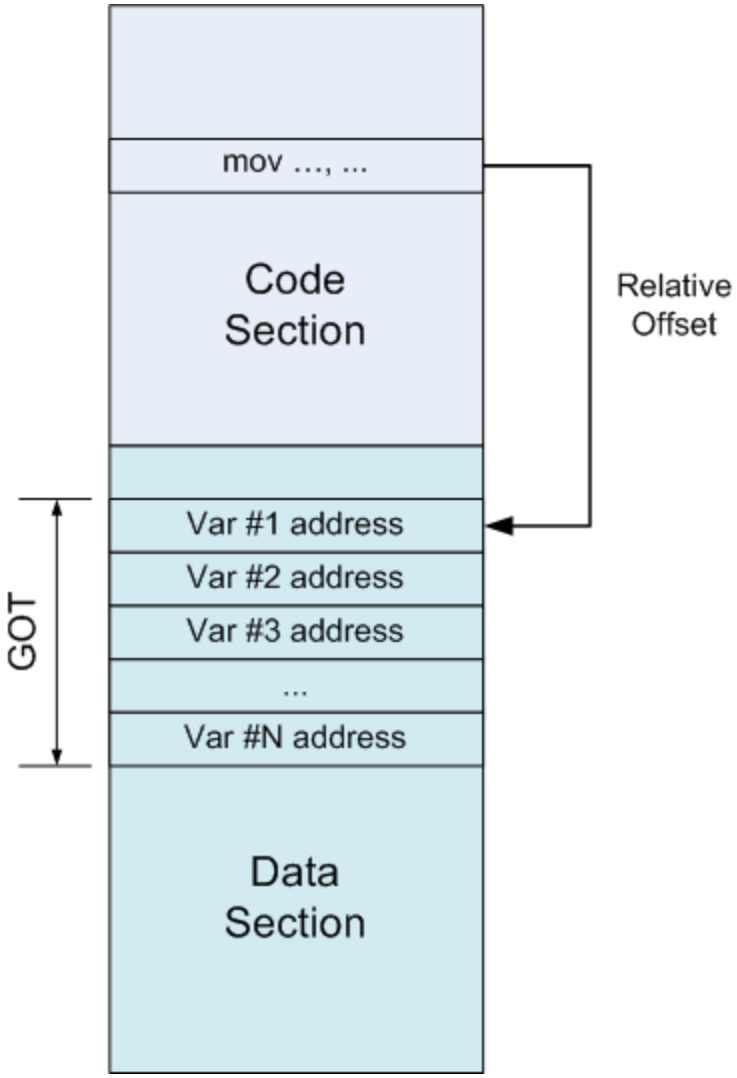
我們都知道動態庫是在運行時綁定的。那麼編譯器是如何找到動態鏈接庫裡面的函數的地址呢？事實上，直到我們第一次調用這個函數，我們並不知道這個函數的地址，這個功能要做延遲綁定lazy bind。因為程序的分支很多，並不是所有的分支都能跑到，想想我們的異常處理，異常處理分支的動態鏈接庫裡面的函數也許永遠跑不到，所以，啟動時解析所有出現過的動態庫裡面的函數是個浪費的辦法，降低性能並且沒有必要。

Global Offset Table（GOT）

在位置無關代碼中，一般不能包含絕對虛擬地址（如共享庫）。當在程序中引用某個共享庫中的符號時，編譯鏈接階段並不知道這個符號的具體位置，只有等到動態鏈接器將所需要的共享庫加載時進內存後，也就是在運行階段，符號的地址才會最終確定。因此，需要有一個數據結構來保存符號的絕對地址，這就是GOT表的作用，GOT表中每項保存程序中引用其它符號的絕對地址。這樣，程序就可以通過引用GOT表來獲得某個符號的地址。

在x86結構中，GOT表的前三項保留，用於保存特殊的數據結構地址，其它的各項保存符號的絕對地址。對於符號的動態解析過程，我們只需要了解的就是第二項和第三項，即GOT[1]和GOT[2]：GOT[1]保存的是一個地址，指向已經加載的共享庫的鍊錶地址；GOT[2]保存的是一個函數的地址，定義如下：GOT[2] = &_dl_runtime_resolve，這個函數的主要作用就是找到某個符號的地址，並把它寫到與此符號相關的GOT項中，然後將控制轉移到目標函數，後面我們會詳細分析。GOT示意如下圖，GOT表slot的數量就是3

+ number of functions to be loaded.



Procedure Linkage Table（PLT）

過程鏈接表（PLT）的作用就是將位置無關的函數調用轉移到絕對地址。在編譯鏈接時，鏈接器並不能控制執行從一個可執行文件或者共享文件中轉移到另一個中（如前所說，這時候函數的地址還不能確定），因此，鏈接器將控制轉移到PLT中的某一項。而PLT通過引用GOT表中的函數的絕對地址，來把控制轉移到實際的函數。

在實際的可執行程序或者共享目標文件中，GOT表在名稱為.got.plt的section中，PLT表在名稱為.plt的section中。

3. 反彙編

我們使用的代碼是：

```
#include

#include

void fun(int a)

{

a++;
```

```
}

int main()

{

fun(1);

int x = rand();

return 0;

}
```

動態庫裡面需要重定位的函數在.got.plt這個段裡面，通過readelf我們可以看到，它一共有六個地址空間，前三個我們已經解釋了。說明該程序預留了三個所需要重新定位的函數。因此用不到的函數是永遠不會被加載的。

```
 [23] .dynamic DYNAMIC 0000000000600e10 00000e10
      00000000000001d0 0000000000000010 WA 8 0 8
 [24] .got PROGBITS 0000000000600fe0 00000fe0
      0000000000000008 0000000000000008 WA 0 0 8
 [25] .got.plt PROGBITS 0000000000600fe8 00000fe8
      0000000000000048 0000000000000008 WA 0 0 8
```

反彙編main函數：

```
(gdb) disas main
Dump of assembler code for function main:
0x0000000000400549 : push %rbp
0x000000000040054a : mov %rsp,%rbp
0x000000000040054d : sub $0x10,%rsp
0x0000000000400551 : mov $0x1,%edi
0x0000000000400556 : callq 0x40053c
0x000000000040055b : callq 0x400440
0x0000000000400560 : mov %eax,-0x4(%rbp)
0x0000000000400563 : mov $0x0,%eax
0x0000000000400568 : leaveq
0x0000000000400569 : retq
End of assembler dump.
```

可以看到其實調用我們自定義的fun**和系統庫函數**rand**形成的彙編差不多，沒有額外的處理。接著向下看**rand**：**

```
(gdb) disas 0x400440
Dump of assembler code for function rand@plt:
0x0000000000400440 : jmpq *0x200bc2(%rip) # 0x601008
<_GLOBAL_OFFSET_TABLE_+32>
0x0000000000400446 : pushq $0x1
0x000000000040044b : jmpq 0x400420
End of assembler dump.
```

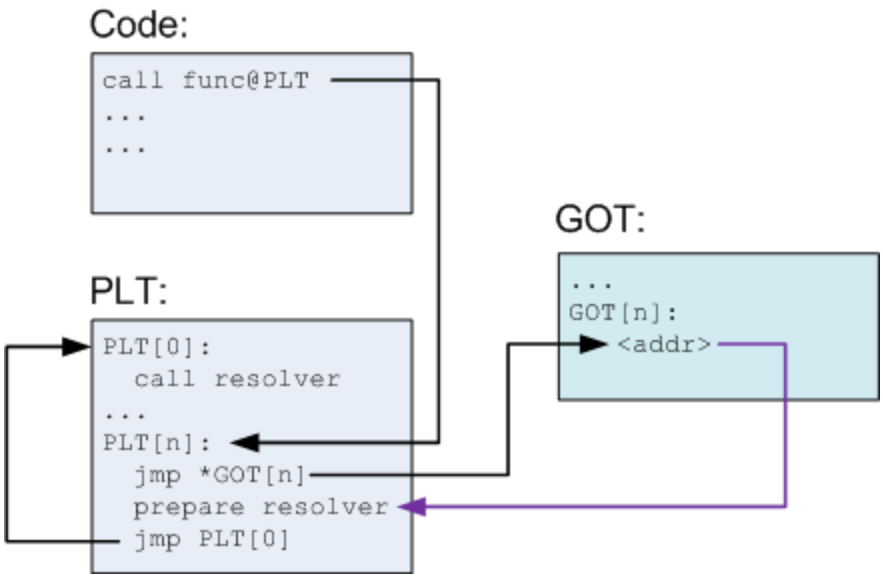
真正有意思的在# 0×601008 <_GLOBAL_OFFSET_TABLE_+32>**。也就是**rand@plt**首先會跳到這裡。我們看一下這裡是什麼：**

(gdb) x 0x601008
0x601008 <_GLOBAL_OFFSET_TABLE_+32>: 0x00400446

接著看0×00400446是什麼：

(gdb) x/5i 0x00400446
0x400446 : pushq $0x1
0x40044b : jmpq 0x400420

可能你注意到了，這裡的處理是和剛才的rand@plt**的**jmpq**一樣。都是將0×1入棧，然後**jmpq　0×400420**。因此這樣就避免了**GOT**表是否為是真實值的檢查：如果是空，那麼去尋址；否則直接調用。**



其實接下來處理的就是調用_dl_runtime_resolve_()函數，該函數最終會尋址到rand的真正地址並且會調用_dl_fixup來將rand的實際地址填入GOT表中。

我們將整個程序執行完，然後看一下0×601008 <_GLOBAL_OFFSET_TABLE_+32>是否已經修改成rand的實際地址：

```
(gdb) x 0x601008
0x601008 <_GLOBAL_OFFSET_TABLE_+32>: 0xf7ab6470
```

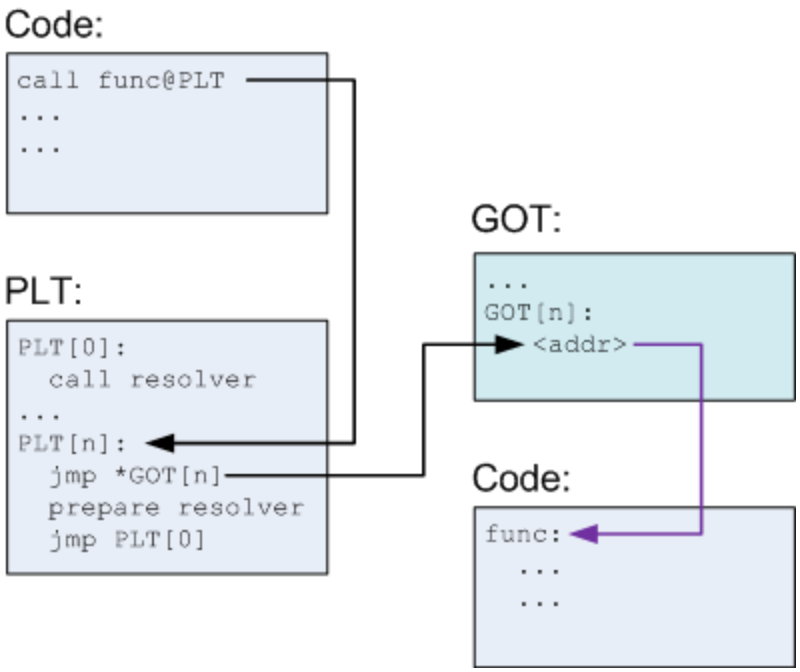
可以看到，rand的地址已經修改為0xf7ab6470了。然後可以通過maps確認一下是否libc load在這個地址：

(gdb) shell cat /proc/`pgrep a.out`/maps
00400000-00401000 r-xp 00000000 08:02 491638 /root/study/got/a.out
00600000-00601000 r--p 00000000 08:02 491638 /root/study/got/a.out
00601000-00602000 rw-p 00001000 08:02 491638 /root/study/got/a.out
7ffff7a80000-7ffff7bd5000 r-xp 00000000 08:02 327685 /lib64/libc-2.11.1.so
7ffff7bd5000-7ffff7dd4000 ---p 00155000 08:02 327685 /lib64/libc-2.11.1.so
7ffff7dd4000-7ffff7dd8000 r--p 00154000 08:02 327685 /lib64/libc-2.11.1.so
7ffff7dd8000-7ffff7dd9000 rw-p 00158000 08:02 327685 /lib64/libc-2.11.1.so
7ffff7dd9000-7ffff7dde000 rw-p 00000000 00:00 0
7ffff7dde000-7ffff7dfd000 r-xp 00000000 08:02 327698 /lib64/ld-2.11.1.so
7ffff7fc4000-7ffff7fc7000 rw-p 00000000 00:00 0
7ffff7ffa000-7ffff7ffb000 rw-p 00000000 00:00 0
7ffff7ffb000-7ffff7ffc000 r-xp 00000000 00:00 0 [vdso]
7ffff7ffc000-7ffff7ffd000 r--p 0001e000 08:02 327698 /lib64/ld-2.11.1.so
7ffff7ffd000-7ffff7ffe000 rw-p 0001f000 08:02 327698 /lib64/ld-2.11.1.so
7ffff7ffe000-7ffff7fff000 rw-p 00000000 00:00 0
7fffffffea000-7ffffffff000 rw-p 00000000 00:00 0 [stack]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]

沒有問題，如我們所分析的那樣：

7ffff7a80000-7ffff7bd5000 r-xp 00000000 08:02 327685 /lib64/libc-2.11.1.so

以後的調用就直接調用庫函數了：



参考資料：

1. http://www.linuxidc.com/Linux/2011-06/37268.htm

2. http://blog.chinaunix.net/uid-24774106-id-3349549.html

3. http://www.linuxidc.com/Linux/2011-06/37268.htm

4.
http://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries/