重定位：

位于 .rel.dyn 中的主要是 R\_ARM\_GLOB\_DAT 类型的重定位项，而位于 .rel.plt 中的主要是 R\_ARM\_JUMP\_SLOT 类型的重定位项。前者用于对数据引用做重定位，而后者用于对函数引用做重定位。除此之外，还看到了 R\_ARM\_RELATIVE 类型的重定位。

实际上对于 R\_ARM\_GLOB\_DAT、R\_ARM\_JUMP\_SLOT 这两种重定位类型来说，只需将符号地址填入被修正的内存即可。而 R\_ARM\_RELATIVE 类型看起来特殊些，它的作用是进行基址重置 (Rebasing) 。

比如指针 p 指向静态变量 a，而静态变量 a 相对于 so 基址的偏移为 A。在编译时，so 的基址为 0，此时 p 的值为 A。而当 so 被装载到内存中时，p 的值就需要加上一个 so 在内存中的基址 base。R\_ARM\_RELATIVE 类型的重定位就是用来干这个的。

1 GOT 表符号偏移

重定位结构体

typedef struct{

Elf32\_Addr r\_offset;

Elf32\_Word r\_info;

}Elf32\_Rel;

// 符号结构体

typedef struct{

Elf32\_Word st\_name;

Elf32\_Addr st\_value;

Elf32\_Word st\_size;

unsigned char st\_info;

unsinged char st\_other;

Elf32\_Half st\_shndx;

}Elf32\_Sym;

获取.rel.plt 每个重定位表所对应的符号的步骤：

1、使用ELF32\_R\_SYM宏（参数为.rel.plt的r\_info）获取符号在.dynsym中的偏移

2、找到对应的dynsym在获取.dynsym的st\_name字段，但这个字段不是字符串，也是一

个偏移，是.dynstr节的偏移

3、通过偏移可以获取到相应的符号

GOT表hook

字符串表 .dynstr

符号表 .dynsym

重定位表 .rel.plt

.got section

\_\_asm\_\_ \_\_volatile\_\_(

STMFD SP!,{R0-R12,LR}\n

);

myfunc();

\_\_asm\_\_ \_\_volatile\_\_(

“LDMFD SP!,{R0-R12}\n”

“LDR LR,=origin\_addr\n”

“LDR LR,[LR]\n”

“BLX LR\n”

“LDMFD SP!,{PC}\n”

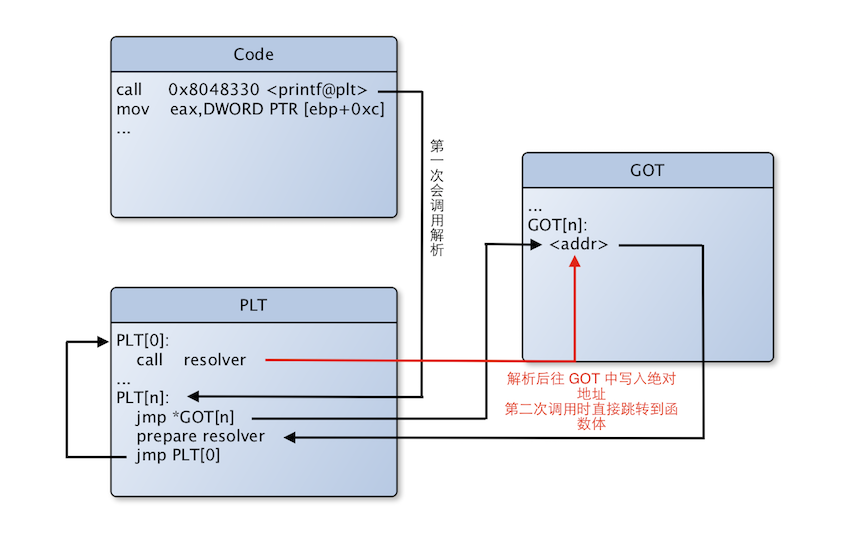
);

全局偏移表和过程链接表

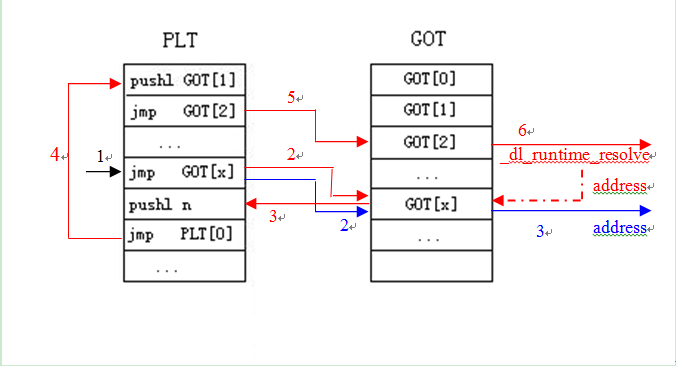
GOT（Global Offset Table）：全局偏移表用于记录在 ELF 文件中所用到的共享库中符号的绝对地址。在程序刚开始运行时，GOT 表项是空的，当符号第一次被调用时会动态解析符号的绝对地址然后转去执行，并将被解析符号的绝对地址记录在 GOT 中，第二次调用同一符号时，由于 GOT 中已经记录了其绝对地址，直接转去执行即可（不用重新解析）。

PLT（Procedure Linkage Table）：过程链接表的作用是将位置无关的符号转移到绝对地址。当一个外部符号被调用时，PLT 去引用 GOT 中的其符号对应的绝对地址，然后转入并执行。

总结来说就是，GOT 保存了程序中所要调用的函数的地址，运行一开时其表项为空，会在运行时实时的更新表项。一个符号调用在第一次时会解析出绝对地址更新到 GOT 中，第二次调用时就直接找到 GOT 表项所存储的函数地址直接调用了。



ld-linux-x86-64.so.2 是一个动态链接库，负责查找程序所使用的函数绝对地址，并将其写入到GOT表中，以供后续调用。其中GOT[0]为空，GOT[1]和GOT[2]用于保存查找的绝对函数地址，GOT[1]保存的是一个地址，指向已经加载的共享库的链表地址；GOT[2]保存的是一个函数的地址，定义如下：GOT[2] = &\_dl\_runtime\_resolve，这个函数的主要作用就是找到某个符号的地址，并把它写到与此符号相关的GOT项中，然后将控制转移到目标函数，而后面的GOT[3]，GOT[4]…都是通过\_dl\_fixup 添加的。



PLT（Procedure Linkage Table）的作用是将位置无关的符号转移到绝对地址。当一个外部符号被调用时，PLT 去引用 GOT 中的其符号对应的绝对地址，然后转入并执行。

GOT（Global Offset Table）：用于记录在 ELF 文件中所用到的共享库中符号的绝对地址。在程序刚开始运行时，GOT 表项是空的，当符号第一次被调用时会动态解析符号的绝对地址然后转去执行，并将被解析符号的绝对地址记录在 GOT 中，第二次调用同一符号时，由于 GOT 中已经记录了其绝对地址，直接转去执行即可（不用重新解析）。

其中PLT 对应.plt section ，而GOT对应 .got.plt ，主要对应函数的绝对地址。通过readelf查看可执行文件，我们发现还存在.got section，这个section主要对应动态链接库中变量的绝对地址。我们还要注意PLT section在代码链接的时候已经存在，存在于代码段中，而GOT存在于数据段中。

ELF的GOT和PLT以及PIC

全局偏移表(GOT)和过程链接表(PLT)

ELF 格式的共享库使用 PIC 技术使代码和数据的引用与地址无关，程序可以被加载到地址空间的任意位置。PIC 在代码中的跳转和分支指令不使用绝对地址。PIC 在 ELF 可执行映像的数据段中建立一个存放所有全局变量指针的全局偏移量表 GOT

对于模块外部引用的全局变量和全局函数，用 GOT 表的表项内容作为地址来间接寻址；对于本模块内的静态变量和静态函数，用 GOT 表的首地址作为一个基准，用相对于该基准的偏移量来引用，因为不论程序被加载到何种地址空间，模块内的静态变量和静态函数与 GOT 的距离是固定的，并且在链接阶段就可知晓其距离的大小。这样，PIC 使用 GOT 来引用变量和函数的绝对地址，把位置独立的引用重定向到绝对位置。

对于 PIC 代码，代码段内不存在重定位项，实际的重定位项只是在数据段的 GOT 表内。共享目标文件中的重定位类型有 R\_386\_RELATIVE、R\_386\_GLOB\_DAT 和 R\_386\_JMP\_SLOT，用于在动态链接器加载映射共享库或者模块运行的时候对指针类型的静态数据、全局变量符号地址和全局函数符号地址进行重定位。

1.2 PLT 表  
过程链接表用于把位置独立的函数调用重定向到绝对位置。通过 PLT 动态链接的程序支持惰性绑定模式。每个动态链接的程序和共享库都有一个 PLT，PLT 表的每一项都是一小段代码，对应于本运行模块要引用的一个全局函数。程序对某个函数的访问都被调整为对 PLT 入口的访问。

每个 PLT 入口项对应一个 GOT 项，执行函数实际上就是跳转到相应 GOT 项存储的地址，该 GOT 项初始值为 PLTn项中的 push 指令地址(即 jmp 的下一条指令，所以第 1 次跳转没有任何作用)，待符号解析完成后存放符号的真正地址。动态链接器在装载映射共享库时在 GOT 里设置 2 个特殊值：在 GOT+4( 即 GOT[1]) 设置动态库映射信息数据结构link\_map 地址；在 GOT+8(即 GOT[2])设置动态链接器符号解  
析函数的地址\_dl\_runtime\_resolve。

PLT 的第 1 个入口 PLT0 是一段访问动态链接器的特殊代码。程序对 PLT 入口的第 1 次访问都转到了 PLT0，最后跳入 GOT[2]存储的地址执行符号解析函数。待完成符号解析后，将符号的实际地址存入相应的 GOT 项，这样以后调用函数时可直接跳到实际的函数地址，不必再执行符号解析函数

操作系统运行程序时，首先将解释器程序即动态链接器ld.so 映射到一个合适的地址，然后启动 ld.so。ld.so 先完成自己的初始化工作，再从可执行文件的动态库依赖表中指定的路径名查找所需要的库，将其加载映射到内存。

Linux用一个全局的库映射信息结构 struct link\_map链表来管理和控制所有动态库的加载，动态库的加载过程实际上是映射库文件到内存中，并填充库映射信息结构添加到链表中的过程。结构 struct link\_map 描述共享目标文件的加载映射信息，是动态链接器在运行时内部使用的一个结构，通过它保持对已装载的库和库中符号的跟踪。  
link\_map 使用双向链接中间件“l\_next”和“l\_prev”链接进程中所有加载的共享库。当动态链接器需要去查找符号的时候，可以向前或向后遍历这个链表，通过访问链表上的每一个库去搜索需要查找的符号。Link\_map 链表的入口由每个可执行映像的全局偏移表的第 2 个入口(GOT[1])指向，查找符号时先从 GOT[1]读取 link\_map 结点地址，然后沿着link-map 结点进行搜索。

动态库的加载映射过程主要分 3 步：  
(1) 动态链接器调用 \_\_mmap 函数对动态库的所有PT\_LOAD 可加载段进行整体映射：

l\_map\_start=(ElfW(Addr))\_\_mmap ((void \*)0, maplength, prot,  
MAP\_COPY | MAP\_FILE, fd, mapoff);

返回值 l\_map\_start 是实际映射的虚拟地址，和段结构成员 p\_vaddr 指定的虚拟地址不一定相同，这对于位置无关代码不会产生影响。但是对于数据段和 link\_map 结构中其它相关的位置描述信息还要进行修正。共享库映射的内存位置关系如图 1，l\_addr 是实际映射地址和原来指定的映射地址的差值，用于其它位置信息的修正，即简单地将原来指定的虚拟地址加上 l\_addr 就可以得到实际加载的虚拟地址

(2)共享文件映射完毕，动态链接器处理共享库的PT\_DYNAMIC 动态段，将各项动态链接信息主要是哈希表、符号表、字符串表、重定位表、PLT 重定位项表等地址填写到 link\_map 的 l\_info 数组结构中。l\_info 是 link\_map 最重要的字段之一，几乎所有与动态链接管理相关的内容都与 l\_info数组有关。动态链接器还要加载处理当前共享库的所有依赖库。

(3)由于实际的映射地址和指定的虚拟地址有可能不同，因此还要对动态库及其依赖库进行重定位。设置动态库的第1 个和第 2 个 GOT 表项：

Elf32\_Addr \*got =  
(Elf32\_Addr \*) lmap->l\_info[DT\_PLTGOT].d\_un.d\_ptr;  
got[1]=lmap;  
got[2]=&\_dl\_runtime\_resolve;

对动态库的所有重定位项进行重定位，在重定位项指定的偏移地址处加上修正值 l\_addr。动态项 DT\_REL 给出了重定位表的地址，DT\_RELSZ 给出重定位表项的数目。  
映射完毕后，动态链接器调用共享库(包括所有相关的依赖库)自备的初始化函数进行初始化。

程序连接表(Procedure Linkage Table)可以使被感染的文件调用外部的函数。这要比修改LD\_PRELOAD环境变量实现调用的重定向优越的多，首先不牵扯到环境变量的修改

程序连接表(PLT)

　　在ELF文件中，全局偏移表(Global Offset Table,GOT)能够把位置无关的地址定位到绝对地址，程序连接表也有类似的作用，它能够把位置无关的函数调用定向到绝对地址。连接编辑器(link editor)不能解决程序从一个可执行文件或者共享库目标到另外一个的执行转移。结果，连接编辑器只能把包含程序转移控制的一些入口安排到程序连接表 (PLT)中。在system V体系中，程序连接表位于共享正文中，但是它们使用私有全局偏移表(private global offset table)中的地址。动态连接器(例如：ld-2.2.2.so)会决定目标的绝对地址并且修改全局偏移表在内存中的影象。因而，动态连接器能够重定向这些入口，而勿需破坏程序正文的位置无关性和共享特性。可执行文件和共享目标文件有各自的程序连接表。

elf的动态连接库是内存位置无关的，就是说你可以把这个库加载到内存的任何位置都没有影响。这就叫做position independent。在编译内存位置无关的动态连接库时，要给编译器加上 -fpic选项，让编译器产生的目标文件是内存位置无关的还会尽量减少对变量引用时使用绝对地址。把库编译成内存位置无关会带来一些花费，编译器会保留一个寄存器来指向全局偏移量表（global offset table (or GOT for short)），这就会导致编译器在优化代码时少了一个寄存器可以使用，但是在最坏的情况下这种性能的减少只有3%，在其他情况下是大大小于3%的。

位置无关 PIC (PE可以吗？)

https://github.com/boyliang/AllHookInOne.[**Git**](http://lib.csdn.net/base/git)

readelf –r libtest\_strlen.so

Relocation section '.rel.dyn' at offset 0xb88 contains 13 entries:

Offset Info Type Sym.Value Sym. Name

00003ea4 00000017 R\_ARM\_RELATIVE

00003fac 00000017 R\_ARM\_RELATIVE

00003fb0 00000017 R\_ARM\_RELATIVE

00003fb4 00000017 R\_ARM\_RELATIVE

00003fb8 00000017 R\_ARM\_RELATIVE

00003fbc 00000017 R\_ARM\_RELATIVE

00003fc4 00000017 R\_ARM\_RELATIVE

00003fc8 00000017 R\_ARM\_RELATIVE

00003fa8 00000415 R\_ARM\_GLOB\_DAT 00000000 strlen

00004004 00000402 R\_ARM\_ABS32 00000000 strlen

00004008 00000402 R\_ARM\_ABS32 00000000 strlen

00003fc0 00000b15 R\_ARM\_GLOB\_DAT 00000000 \_\_gnu\_Unwind\_Find\_exid

00003fcc 00002115 R\_ARM\_GLOB\_DAT 00000000 \_\_cxa\_call\_unexpected

Relocation section '.rel.plt' at offset 0xbf0 contains 9 entries:

Offset Info Type Sym.Value Sym. Name

00003fdc 00000216 R\_ARM\_JUMP\_SLOT 00000000 \_\_cxa\_atexit

00003fe0 00000116 R\_ARM\_JUMP\_SLOT 00000000 \_\_cxa\_finalize

00003fe4 00000416 R\_ARM\_JUMP\_SLOT 00000000 strlen

00003fe8 00000716 R\_ARM\_JUMP\_SLOT 00000000 printf

00003fec 00000b16 R\_ARM\_JUMP\_SLOT 00000000 \_\_gnu\_Unwind\_Find\_exid

00003ff0 00001116 R\_ARM\_JUMP\_SLOT 00000000 abort

00003ff4 00001316 R\_ARM\_JUMP\_SLOT 00000000 memcpy

00003ff8 00001e16 R\_ARM\_JUMP\_SLOT 00000000 \_\_cxa\_begin\_cleanup

00003ffc 00001f16 R\_ARM\_JUMP\_SLOT 00000000 \_\_cxa\_type\_match

00000D10: LDR R4,=(\_GLOBAL\_OFFSET\_TABLE\_ - 0xD1C)

00000D14: ADD R4,PC,R4

00000D18: LDR R3,=(aHelloWorld – 0xD24)

00000D1C: ADD R3,PC,R3 // PC = 0xD24

00000D20:

00000D24:

00000E80: DCD aHelloWorld – 0xD24 0x23D8 – 0xD24 = 0x16B4

.rodata

000023D8 aHelloWorld

.got

00003FA8: 0x4018 (strlen\_ptr) R\_ARM\_GLOB\_DAT 局部函数指针

00003FAC: 0x4004

00003FB0: 0x4008

00003FD0: \_GLOBAL\_OFFSET\_TABLE\_

00003FE4: 0x4018

.data

0x4004: 0x4018

0x4008: 0x4018

当动态链接时，linker会覆盖0x00004004地址的值，指向strlen的真正地址（而不是现在的0x00004018，有点绕）。

**通过全局函数指针的方式调用外部函数，它的重定位类型是R\_ARM\_ABS32，并且位于.rel.dyn节区**。

**因此R\_ARM\_GLOB\_DAT的重定项Offset指向最终调用函数地址的地址（也就是函数指针的指针）**

## [Linux下生成动态链接库是否必须使用 -fPIC](http://www.cnblogs.com/Totems/p/3191761.html)

在 Linux 下制作动态链接库，“标准” 的做法是编译成位置无关代码（Position Independent Code，PIC），然后链接成一个动态链接库。经常遇到的一个问题是 -fPIC 是不是必需，因为好像不加经常也能正常运行，只是创建 .so 的时候会有一个警告。

搜索、试验了一下，答案似乎是这样：

(1) 通常的建议是始终加上 -fPIC 生成位置无关代码；

(2) AMD64 下，必须使用位置无关代码，否则连接失败：

relocation R\_X86\_64\_32S against `a local symbol' can not be used when making a shared object; recompile with -fPIC

(3) IA32 下，连接成功，但有警告：

warning: creating a DT\_TEXTREL in object.

这样的 .so 文件可以完全正常工作。

可执行文件在链接时就知道每一行代码、每一个变量会被放到线性地址空间的什么位置，因此这些地址可以都作为常数写到代码里面。对动态库，这就不行了，这要等到加载时才知道。无非下面两种方法：

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

生成动态库时假定它被加载在地址 0 处。加载时它会被加载到一个地址（base），这时要进行一次重定位（relocation），把代码、数据段中所有的地址加上这个 base 的值。这样代码运行时就能使用正确的地址了。

(2) 位置无关代码（position independent code）：使用 -fPIC 的 Linux SO。

这样的代码本身就能被放到线性地址空间的任意位置，无需修改就能正确执行。通常的方法是获取指令指针（如 IA32 的 EIP 寄存器）的值，加上一个偏移得到全局变量/函数的地址。

PIC vs. relocatable：

(1) PIC 的缺点主要就是代码有可能长一些。例如 IA32，由于不能直接使用 [EIP+constant] 这样的寻址方式，甚至不能直接将 EIP 的值交给其他寄存器，要用到 GOT（global offset table）来定位全局变量和函数。这样导致代码的效率略低。

(2) PIC 的加载速度稍快，因为不需要做重定位。

(3) 多个进程引用同一个 PIC 动态库时，可以共用内存。这一个库在不同进程中的虚拟地址不同，但操作系统显然会把它们映射到同一块物理内存上。**对于可重定位代码，则必须为每个库都在物理内存中复制一份副本，因为需要修改其中的地址。**当然，主流现代操作系统都启用了分页内存机制，这使得重定位时可以使用 COW（copy on write）来节省内存（32 位 Windows 就是这样做的）；然而，页面的粒度还是比较大的（例如 IA32 上是 4KiB），至少对于代码段来说能节省的相当有限。

注：对于 AMD64，由于 AMD64 实现了 [RIP+constant] 的寻址方式，第 (1) 点不成立。

这样，把动态库编译成 PIC 只有好处没有坏处，因而 Linux AMD64 要求用于生成动态库的目标文件必须使用 -fPIC 编译也合情合理了。

共享库的一种很常用的实现方式就是位置无关代码(position independent  
code)，该方式生成的共享库，其代码部分在动态载入并链接到内存中时是不会有任何修改的，  
这些代码放到内存的任何位置都可以正常工作，所以叫做与位置无关的代码。

这里我们关注位置无关代码(PIC)是如何实现的。毕竟共享库映射到进程内存空间时，其  
位置是不固定的，程序代码是如何知道共享库中函数的具体地址的，共享库中函数之间  
相互调用时又是如何确定具体内存地址的，共享库中的全局变量(或者static)变量地址  
又是如何确定的？

这些问题其实很简单，魔力之源就是全局偏移表(Global Offset Table)，它被放置  
在.so文件的.GOT section中。当调用函数或者引用数据(.so中的数据)时，不是  
直接使用函数或者数据的绝对地址，而是使用GOT中相关索引中保存的地址。也就是说所有  
函数、全局变量的绝对内存地址都放到GOT中，GOT其实就是个unsigned int数组。

UPX 加壳原理

<https://www.pysol.org:4443/hg/upx.hg/tags>