

第3节：延迟绑定与GOT表劫持

- 动态链接与延迟绑定
- GOT表的概念
- 在gdb中观察延迟绑定
- 延迟绑定的过程
- GOT表劫持攻击及实例

动态链接(Dynamic Linking)

- 动态链接
 - 一种运行时才会加载和链接程序所依赖的共享库的技术
 - Linux最常见的共享库是libc
- 重定位(Relocations)
 - 指二进制文件中的待填充项
 - 链接器在链接时填充, 例如链接多个目标文件时, 修正相互引用的函数、变量地址
 - 动态链接器在运行时填充, 例如动态解析库函数(例如 printf)
- 动态链接中的延迟绑定
 - 外部函数的地址在运行时才会确定
 - 外部函数符号通常在首次调用时才会被解析
 - 外部变量不使用延迟绑定机制

GOT表(Global Offset Table)

- GOT表常常用于存放外部库函数地址(或外部变量)
- GOT表项初始状态指向一段PLT(过程链接表, Procedure Linkage Table)代码
- 当库函数被首次调用, 真正的函数地址会被解析并填入相应的GOT表项
- 每个外部函数均有一段PLT(过程链接表, Procedure Linkage Table)代码, 用于跳转到相应GOT表项中存储的地址

在 gdb 中观察延迟绑定

hello.c:

```
void main() {  
    puts("Hello World!");  
}
```

我们使用 gdb 调试 hello 程序，在调用动态库函数 puts() 之前下一个断点，随后观察整个调用过程。

```
$ gdb ./hello  
(gdb) disassemble main  
Dump of assembler code for function main:  
0x0804840b <+0>:    lea    0x4(%esp),%ecx  
0x0804840f <+4>:    and    $0xffffffff0,%esp  
0x08048412 <+7>:    pushl  -0x4(%ecx)  
0x08048415 <+10>:   push   %ebp  
0x08048416 <+11>:   mov    %esp,%ebp  
0x08048418 <+13>:   push   %ecx  
0x08048419 <+14>:   sub    $0x4,%esp  
0x0804841c <+17>:   sub    $0xc,%esp  
0x0804841f <+20>:   push   $0x80484c0  
0x08048424 <+25>:   call   0x80482e0 <puts@plt>  
0x08048429 <+30>:   add    $0x10,%esp  
0x0804842c <+33>:   nop  
0x0804842d <+34>:   mov    -0x4(%ebp),%ecx  
0x08048430 <+37>:   leave  
0x08048431 <+38>:   lea    -0x4(%ecx),%esp  
0x08048434 <+41>:   ret  
End of assembler dump.  
(gdb) break *0x08048424  
Breakpoint 1 at 0x8048424: file hello.c, line 2.  
(gdb) run
```

在 gdb 中观察延迟绑定

```
(gdb) display /3i $eip      设置自动显示后续3条指令
1: x/3i $eip
=> 0x8048424 <main+25>: call    0x80482e0 <puts@plt>
    0x8048429 <main+30>: add     $0x10,%esp
    0x804842c <main+33>: nop
(gdb) stepi                 单步执行下一条指令
0x80482e0 in puts@plt ()
1: x/3i $eip
=> 0x80482e0 <puts@plt>:      jmp     *0x804a00c
    0x80482e6 <puts@plt+6>:   push    $0x0
    0x80482eb <puts@plt+11>:  jmp     0x80482d0
(gdb) x/wx 0x804a00c
0x804a00c:      0x080482e6
puts的GOT表项初始
状态指向puts@plt+6
```

在gdb中, 使用display命令可以设置每次单步执行后自动显示的内容, 此处我们设置为显示后续三条指令。

单步执行call puts@plt, 跳转到puts函数的PLT代码, 第一条指令是jmp *puts@got, 即puts的GOT表项(左边图中GOT表项位于0x804a00c)中包含的地址, 初始状态为0x80482e6, 指向puts@plt+6

在 gdb 中观察延迟绑定

```
(gdb) stepi
0x080482e6 in puts@plt ()
1: x/3i $eip
=> 0x080482e6 <puts@plt+6>:
    0x080482eb <puts@plt+11>:
    ...
(gdb) stepi 2
0x080482d0 in ?? ()
1: x/3i $eip
=> 0x080482d0:
    0x080482d6:
    ...
(gdb) stepi 2
0xf7fedaa0 in _dl_runtime_resolve () from /lib/ld-linux.so.2
1: x/3i $eip
=> 0xf7fedaa0 <_dl_runtime_resolve>:  push    %eax
    0xf7fedaa1 <_dl_runtime_resolve+1>: push    %ecx
    0xf7fedaa2 <_dl_runtime_resolve+2>: push    %edx
```

第一调用puts(), GOT表项初始化为puts@plt+6
跳转到puts@plt+6

push	\$0x0
jmp	0x080482d0

传入第一个参数0,
跳转到0x080482d0
(一般称为PLT0)

pushl	0x804a004
jmp	*0x804a008

传入第一个参数0x804a004,
此处存的是link_map
跳转到*0x804a008(_dl_runtime_resolve)

GOT表项初始化为
puts@plt+6, 第一次调用puts()
函数会执行
_dl_runtime_resolve, 执行完毕
后GOT表项内的指就会填充正
确的puts()函数地址。

后续所有对puts函数的调用都
无需再次解析, 可以直接找到
相应代码。

在 gdb 中观察延迟绑定

```
(gdb) finish      执行完_dl_runtime_resolve, puts()函数被正确解析并执行
Run till exit from #0  0xf7fedaa0 in _dl_runtime_resolve () from
/lib/ld-linux.so.2
Hello World!
0x08048429 in main () at hello.c:2
2          puts("Hello World!");
1: x/3i $eip      返回到puts()函数下一条指令
=> 0x08048429 <main+30>: add      $0x10,%esp      , 我们再次查看puts函数的
      0x0804842c <main+33>: nop
      0x0804842d <main+34>: mov      -0x4(%ebp),%ecx      GOT表项
(gdb) x/wx 0x804a00c
0x804a00c:      0xf7e44300      puts()函数的GOT表项就在地址
(gdb) x/i 0xf7e44300      0x804a00c处, 可以看到已经被解析
      0xf7e44300 <puts>:      push      %ebp      为0xf7e44300, 与libc中的puts函数
(gdb) p puts      地址一致
$1 = {<text variable, no debug info>} 0xf7e44300 <puts>
```

GOT表项初始化为puts@plt+6, 首先跳转到这里。

puts@plt+6处的代码会push第一个参数0, 然后跳到0x80482d0(称为PLT0)

PLT0处的代码会push第一个参数0x804a004(此处存的是link_map), 然后跳到*0x804a008, 实际是_dl_runtime_resolve函数, 一个用来解析动态链接函数的函数。

GOT表位于.got和.got.plt Section

```
$ readelf -S hello
```

```
There are 36 section headers, starting at offset 0x1960:
```

```
Section Headers:
```

```
...
```

```
[22] .dynamic          DYNAMIC          08049f14 000f14 0000e8 08  WA  6  0  4
```

```
[23] .got               PROGBITS         08049ffc 000ffc 000004 04  WA  0  0  4
```

```
[24] .got.plt           PROGBITS         0804a000 001000 000014 04  WA  0  0  4
```

```
[25] .data              PROGBITS         0804a014 001014 000008 00  WA  0  0  4
```

```
...
```

- .got Section中存放外部全局变量的GOT表, 例如stdin/stdout/stderr, 非延迟绑定
- .got.plt Section中存放外部函数的GOT表, 例如printf, 采用延迟绑定

GOT表(Global Offset Table)

```
$ gdb hello
Reading symbols from hello...done.
(gdb) b *0x08048424
Breakpoint 1 at 0x08048424: file hello.c, line 2.
(gdb) r
Starting program: /tmp/hello

Breakpoint 1, 0x08048424 in main () at hello.c:2
2          puts("Hello World!");
(gdb) x/4wx 0x804a000
0x804a000:      0x08049f14      0xf7ffd8f8
0x804a008:      0xf7fedaa0      0x080482e6
(gdb) ni
Hello World!
0x08048429      2          puts("Hello World!");
(gdb) x/4wx 0x804a000
0x804a000:      0x08049f14      0xf7ffd8f8
0x804a008:      0xf7fedaa0      0xf7e44300
```

.got.plt(解析前)

.dynamic section地址	0x804a000
link_map地址	0x804a004
_dl_runtime_resolve	0x804a008
puts@plt + 6	

.got.plt(解析后)

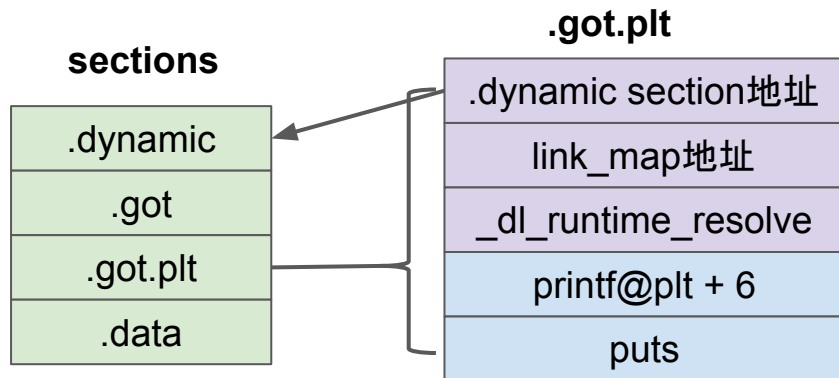
.dynamic section地址
link_map地址
_dl_runtime_resolve
puts



.got.plt前三项有特殊含义, 第四项开始保存引用的各个外部函数的GOT表项:

- 第一项保存的是.dynamic section的地址
- 第二项保存的是link_map结构地址
- 第三项保存了_dl_runtime_resolve函数的地址

GOT表(Global Offset Table)



- .dynamic section
 - 为动态链接提供信息, 例如符号表、字符串表
- link_map
 - 一个链表, 包含所有加载的共享库信息
- _dl_runtime_resolve
 - 位于loader中, 用于解析外部函数符号的函数
 - 解析完成后会直接执行该函数

.plt section

```
$ readelf -S hello
```

```
There are 36 section headers, starting at offset 0x1960:
```

```
Section Headers:
```

```
...  
[12] .plt          PROGBITS          080482d0 0002d0 000030 04  AX  0    0 16  
...
```

- .plt Section中存放所有外部函数对应的PLT代码

.plt section

(gdb) x/10i 0x80482d0

```
0x80482d0:  pushl  0x804a004
0x80482d6:  jmp     *0x804a008
0x80482dc:  add     %al, (%eax)
0x80482de:  add     %al, (%eax)
```

PLT0, push link_map, 跳转到
_dl_runtime_resolve

```
=> 0x80482e0 <puts@plt>:      jmp     *0x804a00c
0x80482e6 <puts@plt+6>:      push    $0x0
0x80482eb <puts@plt+11>:     jmp     0x80482d0
```

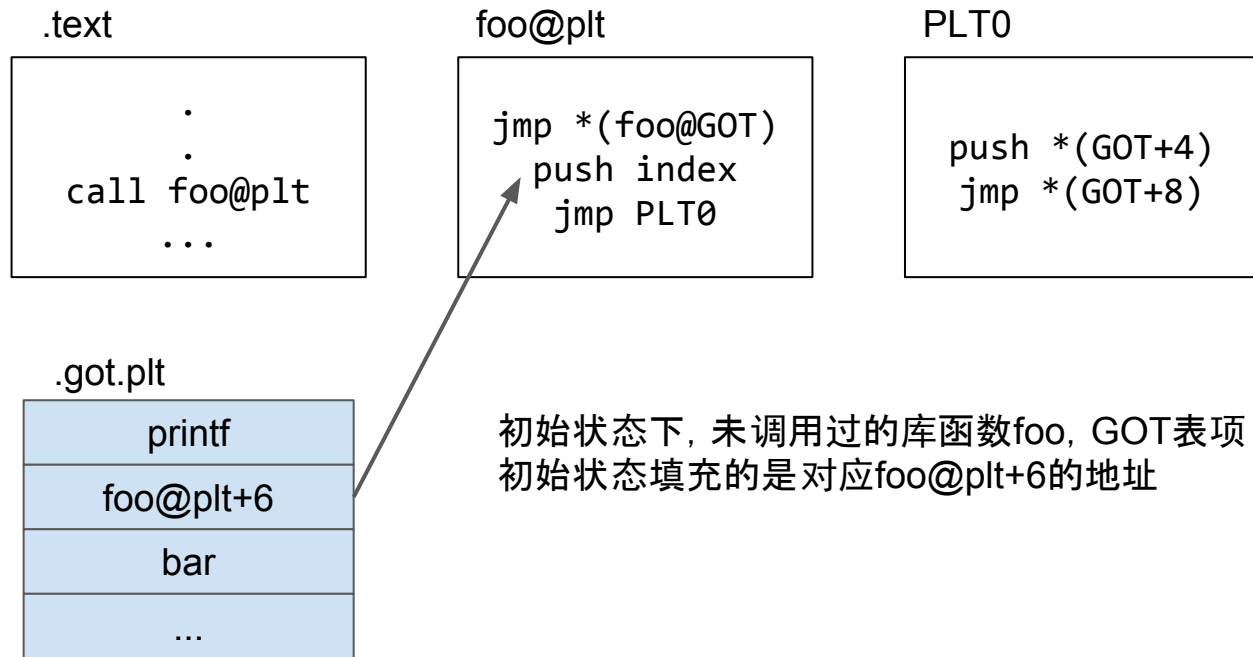
puts的PLT代码

0x804a00c是puts@got, 初始状态指向puts@plt+6,
第一次调用时跳转至PLT0进行符号解析

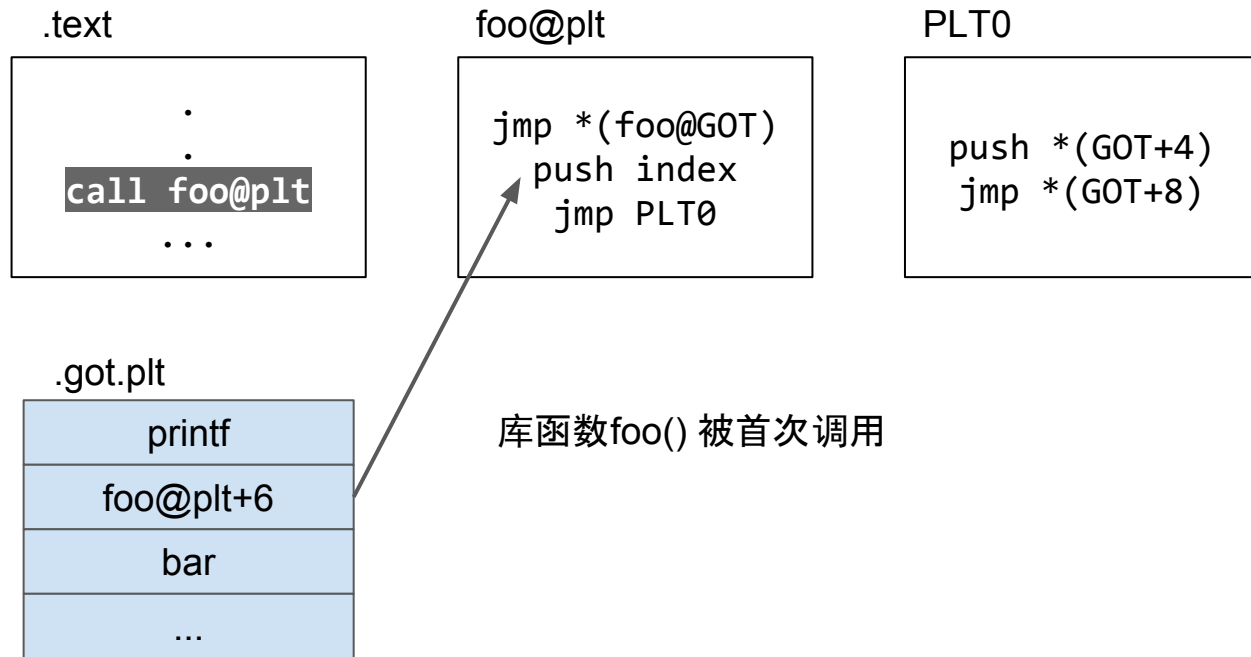
```
0x80482f0 <__libc_start_main@plt>:  jmp     *0x804a010
0x80482f6 <__libc_start_main@plt+6>: push    $0x8
0x80482fb <__libc_start_main@plt+11>: jmp     0x80482d0
```

__libc_start_main的PLT代码, 类
似puts的PLT代码

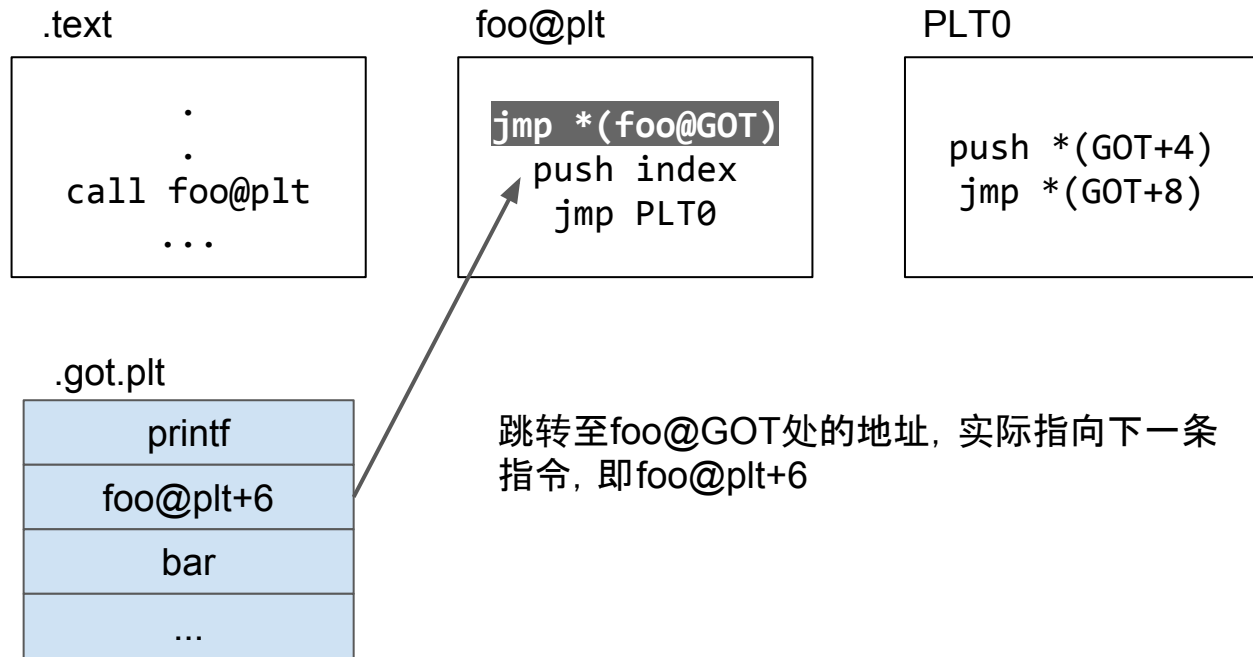
延迟绑定(Lazy Binding)



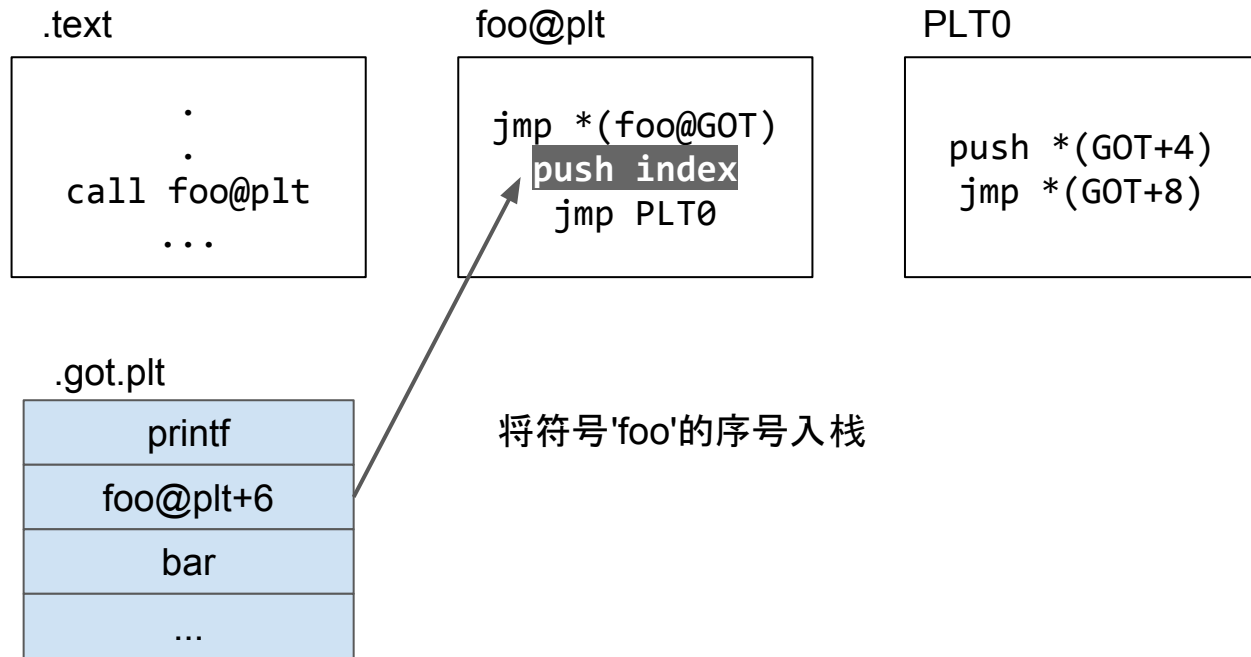
延迟绑定(Lazy Binding)



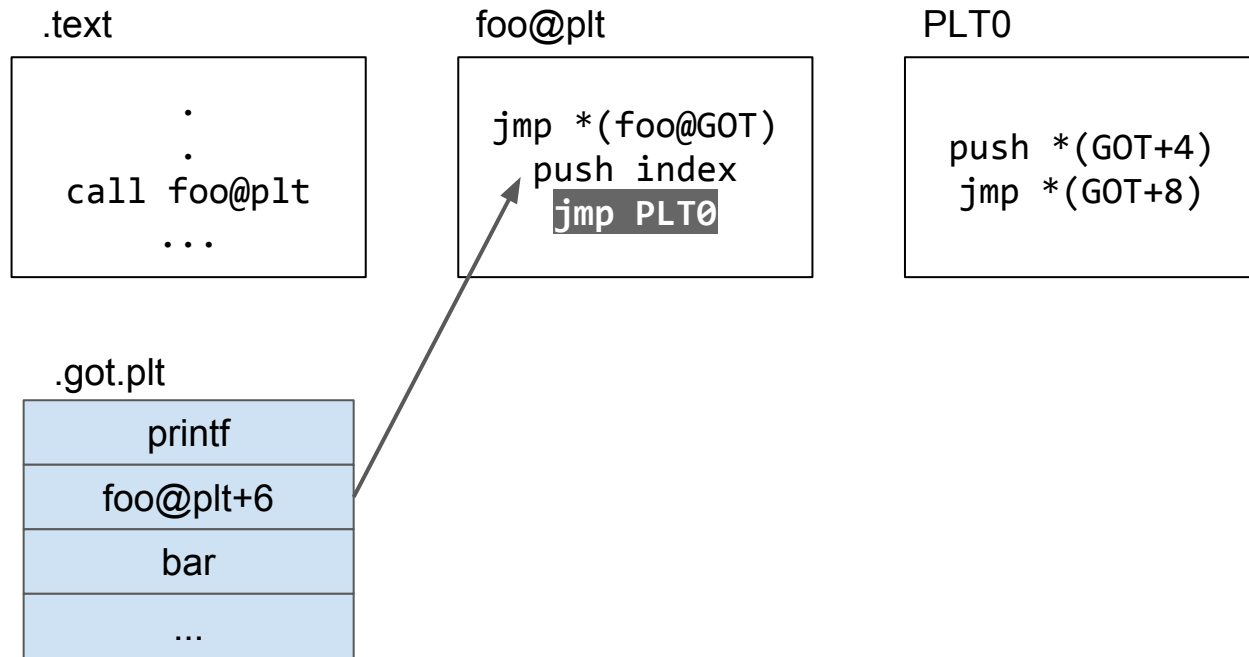
延迟绑定(Lazy Binding)



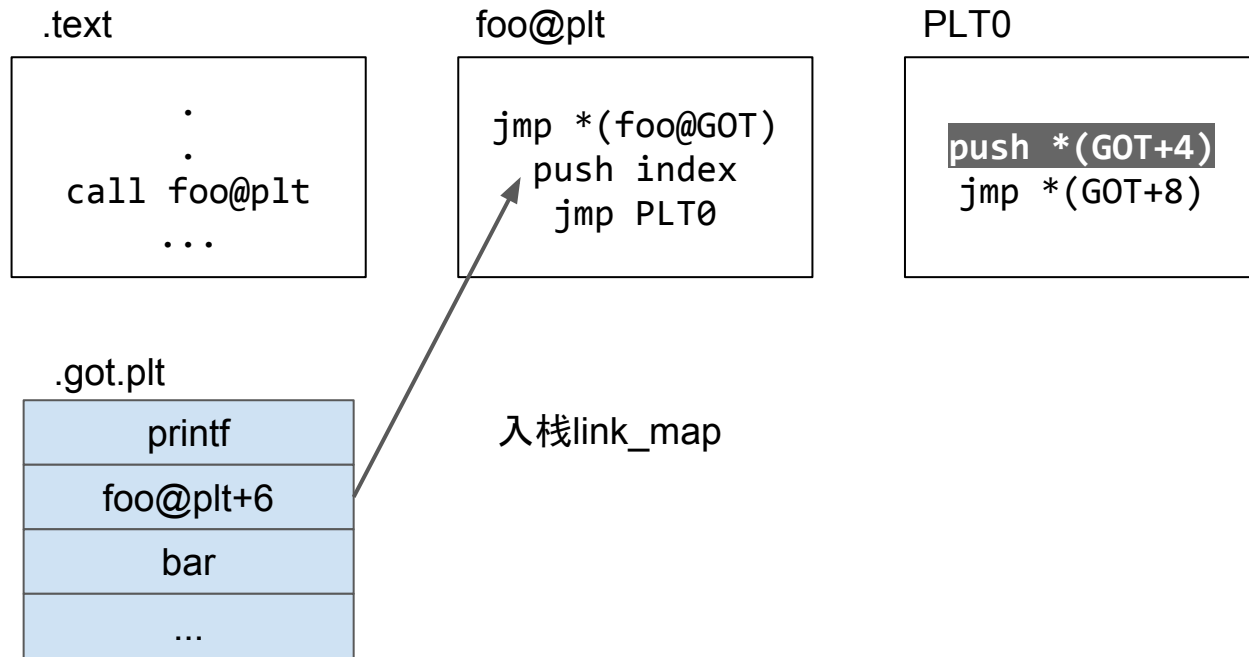
延迟绑定(Lazy Binding)



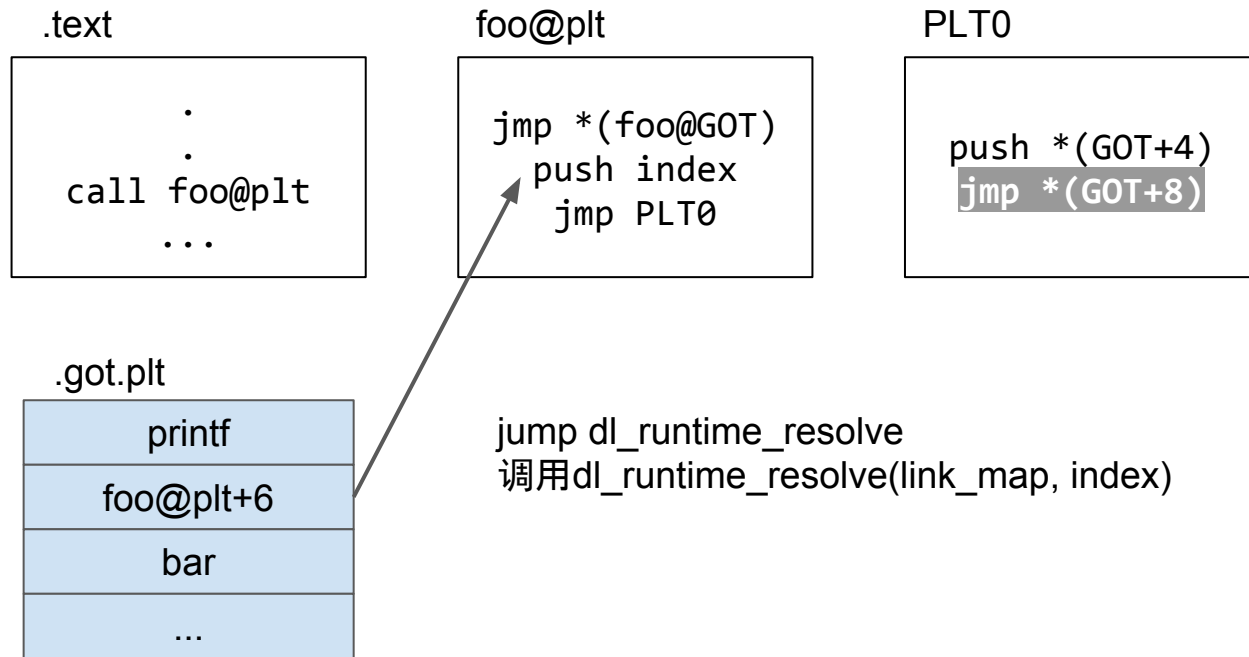
延迟绑定 (Lazy Binding)



延迟绑定(Lazy Binding)



延迟绑定(Lazy Binding)



延迟绑定(Lazy Binding)

.text

```
·  
·  
call foo@plt  
...
```

dl_resolve

```
...  
call _fix_up  
...  
ret 0xc
```

.got.plt

printf
foo@plt+6
bar
...

_fix_up函数解析符号

延迟绑定(Lazy Binding)

.text

```
·  
·  
call foo@plt  
...
```

dl_resolve

```
...  
call _fix_up  
...  
ret 0xc
```

.got.plt

printf
foo
bar
...

foo()函数真实地址已填入GOT表

延迟绑定(Lazy Binding)

.text

```
·  
·  
call foo@plt  
...
```

foo@plt

```
jmp *(foo@GOT)  
push index  
jmp PLT0
```

PLT0

```
push *(GOT+4)  
jmp *(GOT+8)
```

.got.plt

printf
foo
bar
...

库函数foo() 被第二次调用

延迟绑定(Lazy Binding)

.text

```
·  
·  
call foo@plt  
...
```

foo@plt

```
jmp *(foo@GOT)  
push index  
jmp PLT0
```

PLT0

```
push *(GOT+4)  
jmp *(GOT+8)
```

.got.plt

printf
foo
bar
...

直接跳转至库函数开始执行

查找 GOT 表项

```
$ objdump -R ropasaurusrex
```

```
ropasaurusrex:      file format elf32-i386
```

DYNAMIC RELOCATION RECORDS

OFFSET	TYPE	VALUE
08049600	R_386_GLOB_DAT	__gmon_start__
08049610	R_386_JUMP_SLOT	__gmon_start__
08049614	R_386_JUMP_SLOT	write@GLIBC_2.0
08049618	R_386_JUMP_SLOT	__libc_start_main@GLIBC_2.0
0804961c	R_386_JUMP_SLOT	read@GLIBC_2.0

GOT 表劫持

```
$ readelf -S ropasaurusrex
There are 28 section headers, starting at offset 0x724:

Section Headers:
  [Nr] Name                Type           Addr          Off          Size      ES Flg Lk Inf Al
  ...
  [23] .got.plt              PROGBITS       08049604 000604 00001c 04  WA  0  0  4
  ...
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  0 (extra OS processing required) o (OS specific), p (processor specific)
```

- 延迟绑定机制要求GOT表必须可写
- 内存漏洞可导致GOT表项被改写, 从而劫持PC

GOT 表劫持

.text

```
·  
·  
call foo@plt  
...
```

foo@plt

```
jmp *(foo@GOT)  
push index  
jmp PLT0
```

PLT0

```
push *(GOT+4)  
jmp *(GOT+8)
```

.got.plt

printf
foo
bar
...

程序调用外部函数foo()

GOT 表劫持

.text

```
·  
·  
call foo@plt  
...
```

foo@plt

```
jmp *(foo@GOT)  
push index  
jmp PLT0
```

PLT0

```
push *(GOT+4)  
jmp *(GOT+8)
```

.got.plt

printf
system
bar
...

foo()函数的GOT表项被改成了system()函数的地址。

GOT 表劫持

.text

```
·  
·  
call foo@plt  
...
```

foo@plt

```
jmp *(foo@GOT)  
push index  
jmp PLT0
```

PLT0

```
push *(GOT+4)  
jmp *(GOT+8)
```

.got.plt

printf
system
bar
...

执行foo()时, system()函数被调用。

GOT 表劫持案例

got_hijacking.c:

```
#include <stdlib.h>
#include <stdio.h>
```

```
void win() {
    puts("You Win!");
}
```

```
void main() {
    unsigned int addr, value;
    scanf("%x=%x", &addr, &value);
    *(unsigned int *)addr = value;
    printf("set %x=%x\n", addr, value);
}
```

编译: gcc got_hijacking.c -m32 -o got_hijacking

程序允许修改任意地址的4字节, 如何执行win函数呢?

main函数在修改内存后调用了printf函数, 因此可以考虑修改printf的GOT表项, 将其劫持到win()函数。

GOT 表劫持案例

```
$ objdump -R got_hijacking | grep printf
0804a00c R_386_JUMP_SLOT  printf@GLIBC_2.0
$ objdump -d got_hijacking | grep win
0804848b <win>:
$ ./got_hijacking
0804a00c=0804848b
You Win!
```

查询printf@GOT表地址

objdump -d反汇编, 查询win()函数地址

劫持printf的GOT表项为win()函数地址