Return to dl-resolve

ret2lib without information leak

angelboy

先複習一下GOT 及 lazy binding 的機制

Global Offset Table

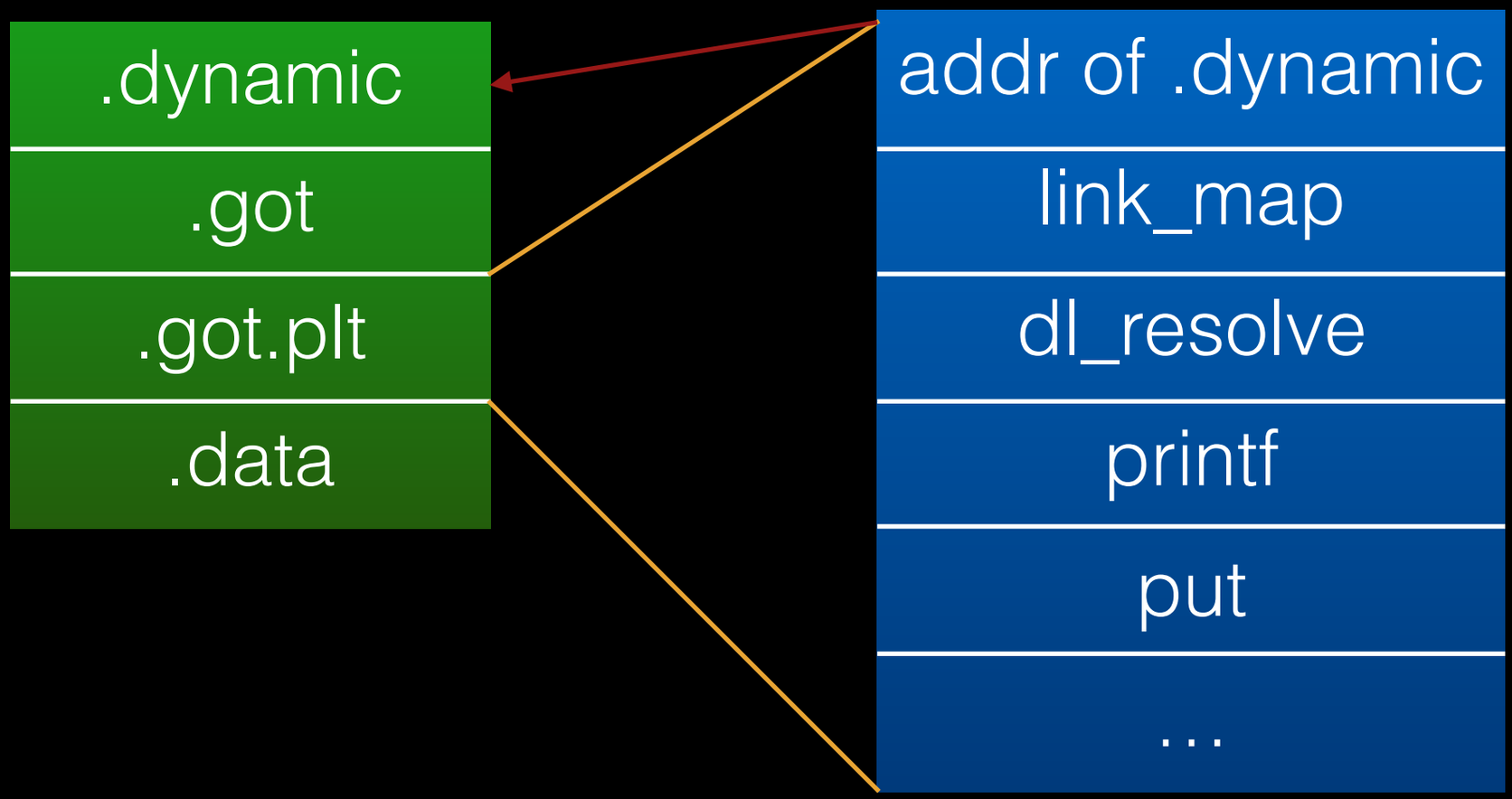
- 分成兩部分
  - .got
    - 保存全域變數引用位置
  - .got.plt
    - 保存函式引用位置

Global Offset Table

- .got.plt
  - 前三項有特別用途
    - address of .dynamic
    - link_map
      - 一個將有引用到的 library 所串成的 linked list
    - dl_runtime_resolve
- 後面則是程式中 .so 函式引用位置

Global Offset Table

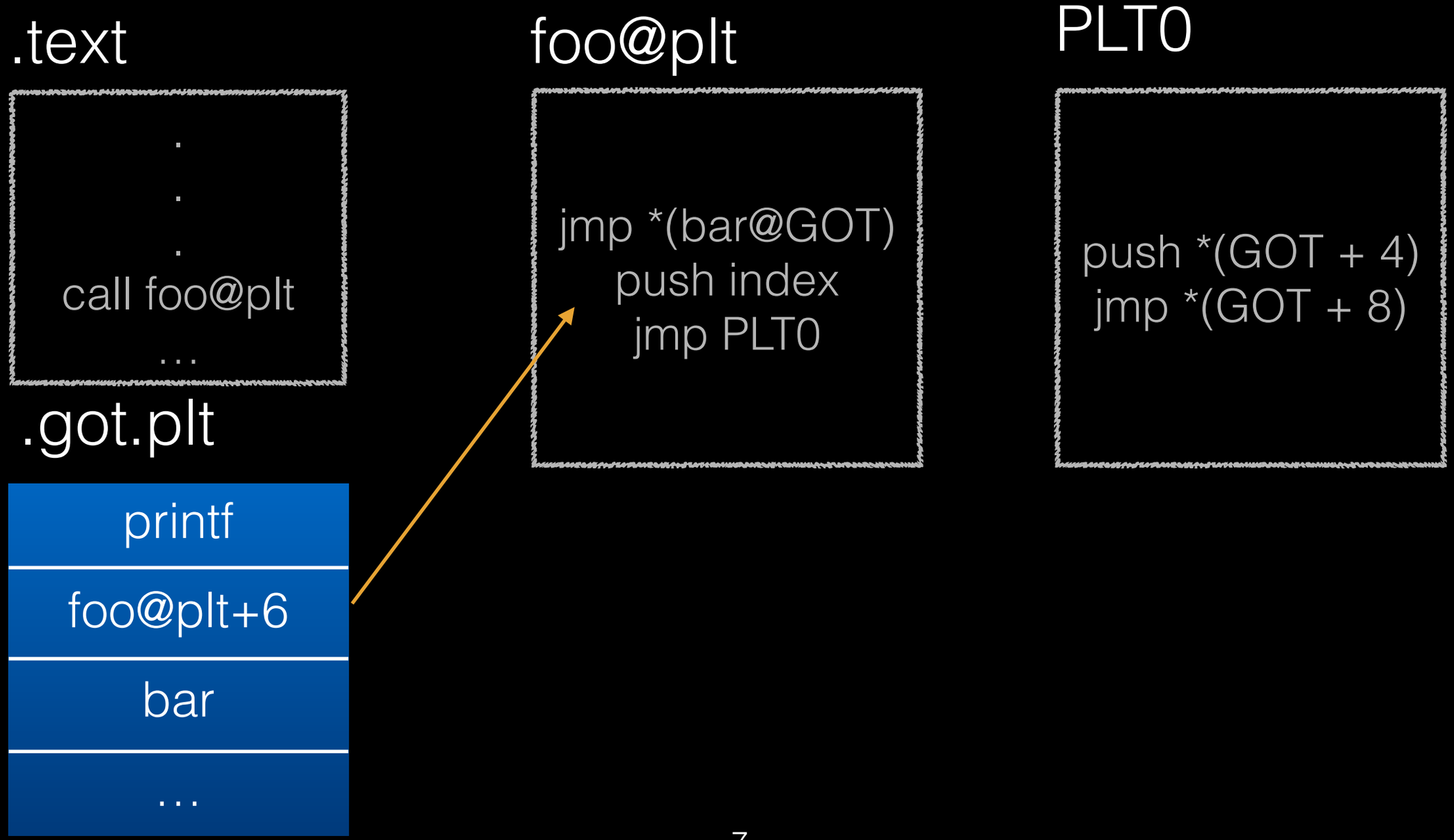
- layout



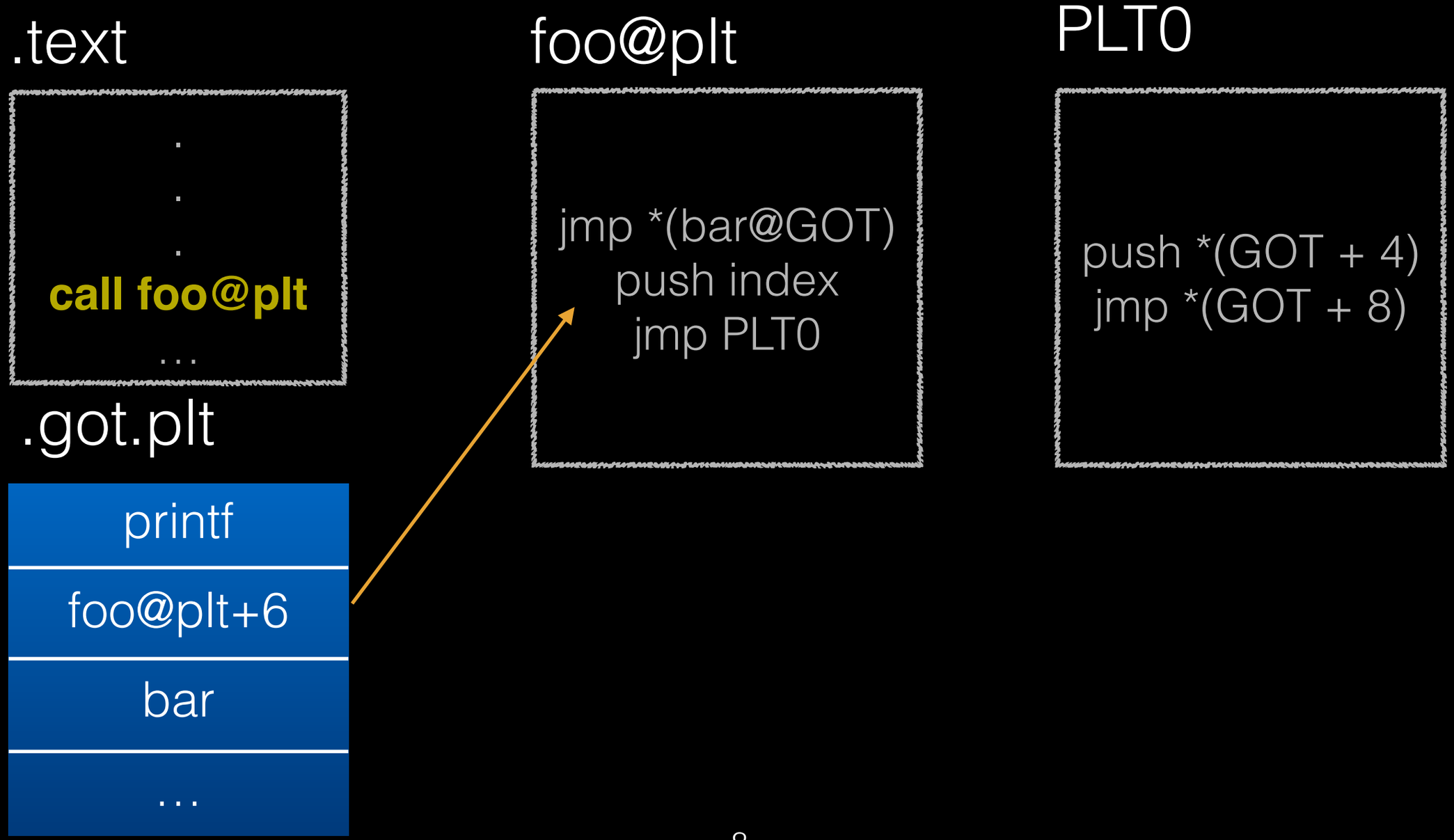
Lazy binding

- 程式在執行過程中，有些 library 的函式可能到結束都不會執行到
- 所以 ELF 採取 Lazy binding 的機制，在第一次 call library 函式時，才會去尋找函式真正的位置進行 binding

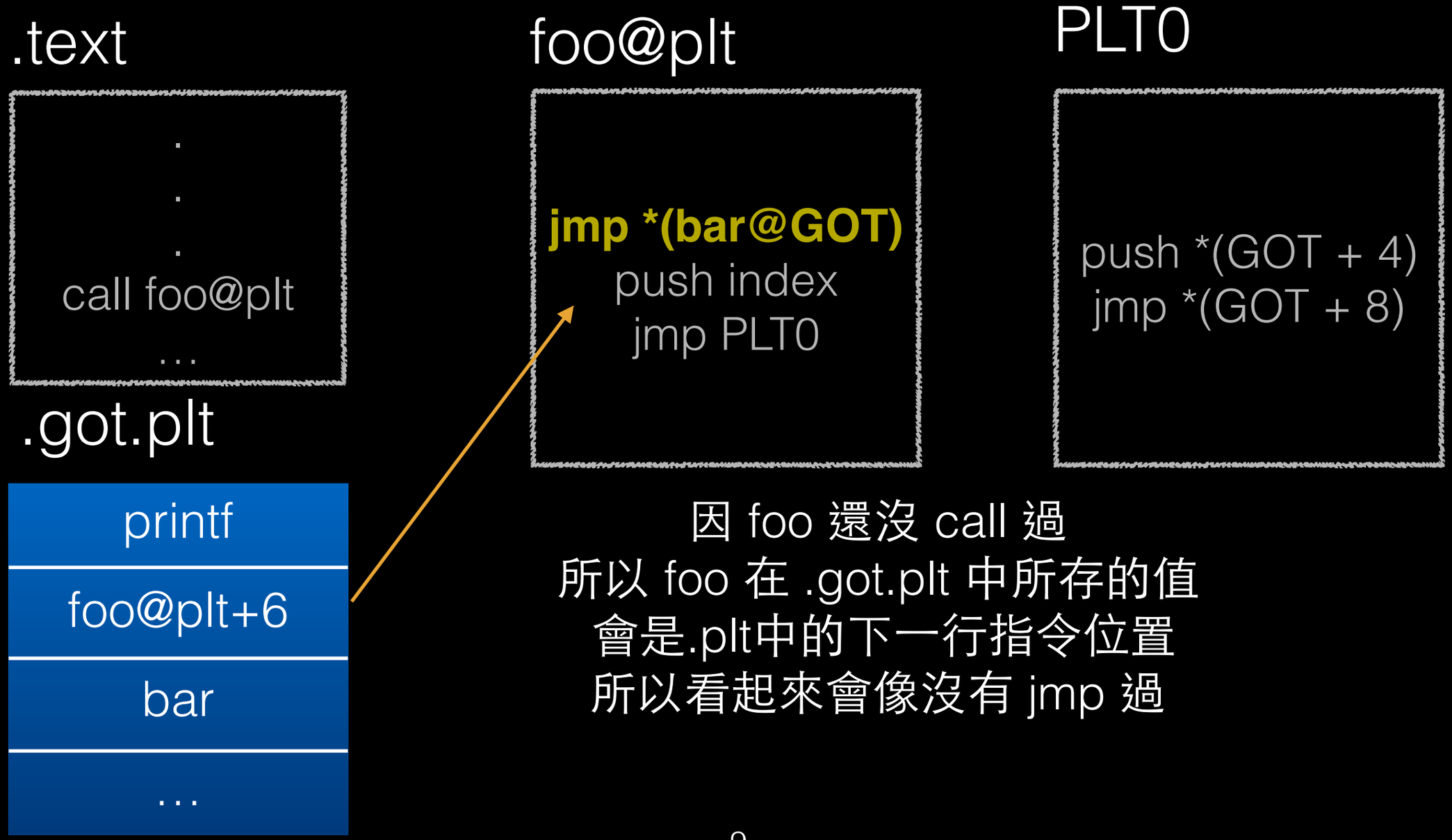
Lazy binding



Lazy binding

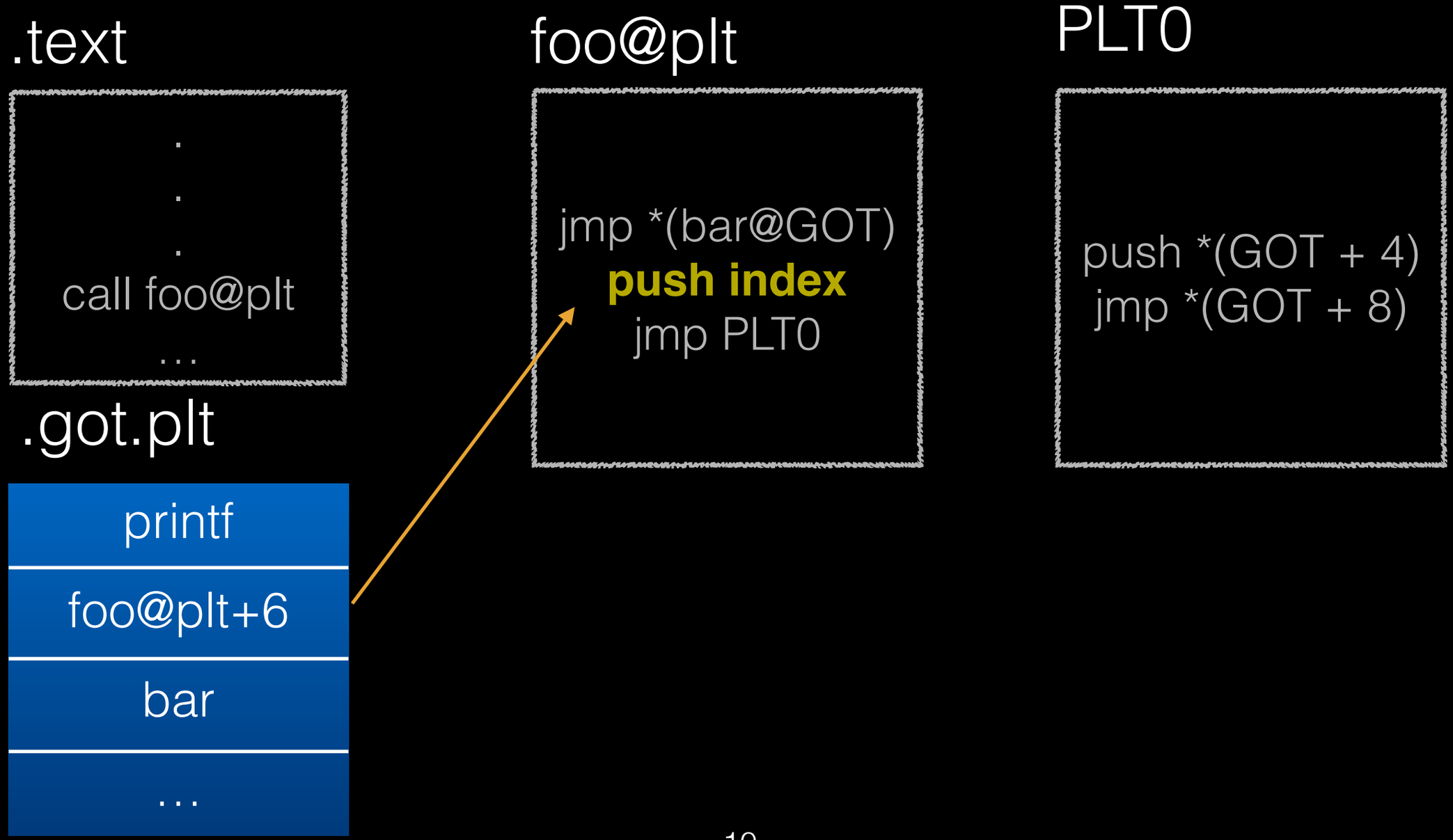


Lazy binding

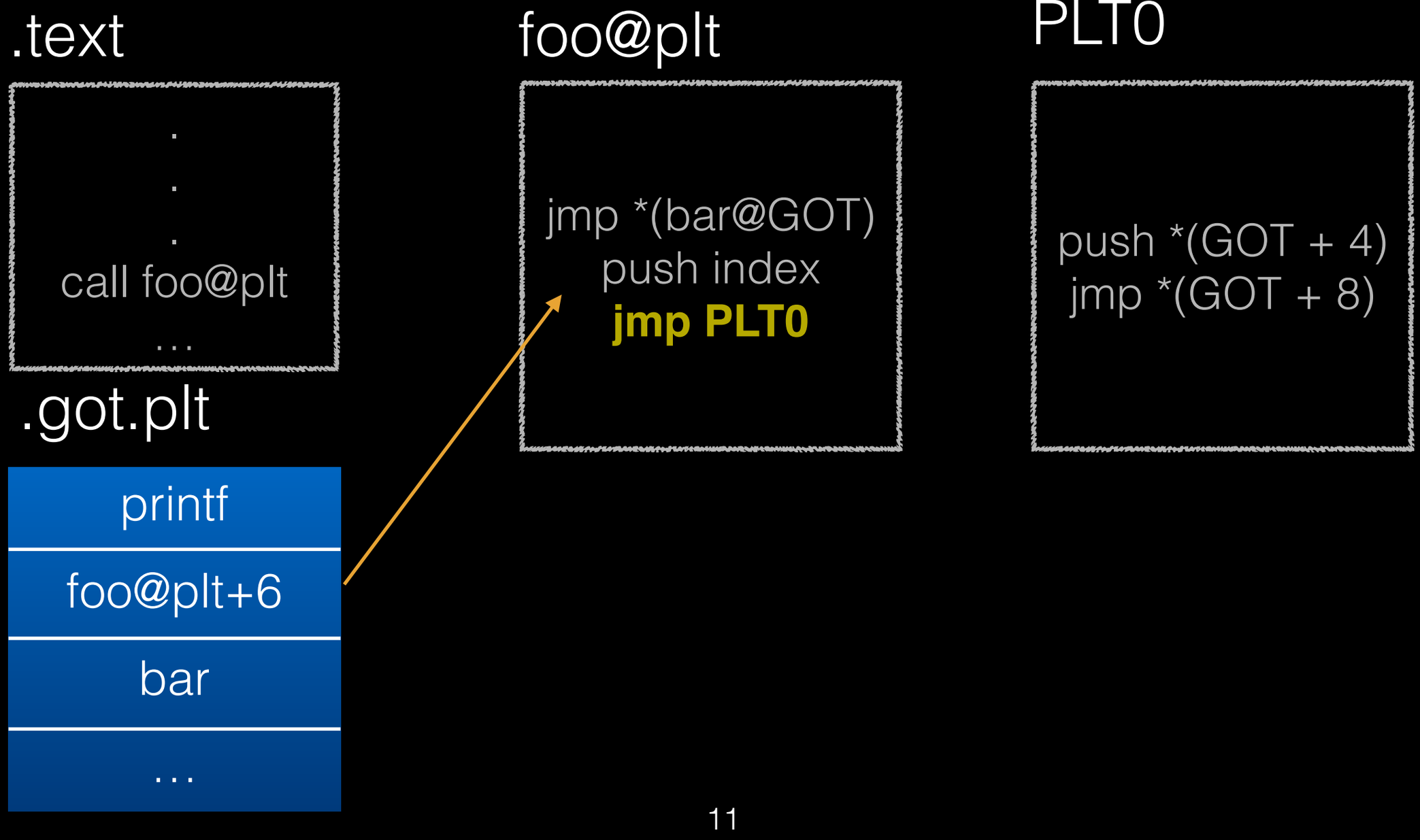


因 foo 還沒 call 過
所以 foo 在 .got.plt 中所存的值
會是.plt中的下一行指令位置
所以看起來會像沒有 jmp 過

Lazy binding



Lazy binding



Lazy binding

.text

.
.
call foo@plt
…

.got.plt

printf
foo@plt+6
bar
…

foo@plt

jmp *(bar@GOT)
push index
jmp PLT0

PLT0

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

push link_map

Lazy binding

.text

.
.
call foo@plt
…

.got.plt

printf
foo@plt+6
bar
…

foo@plt

jmp *(bar@GOT)
push index
jmp PLT0

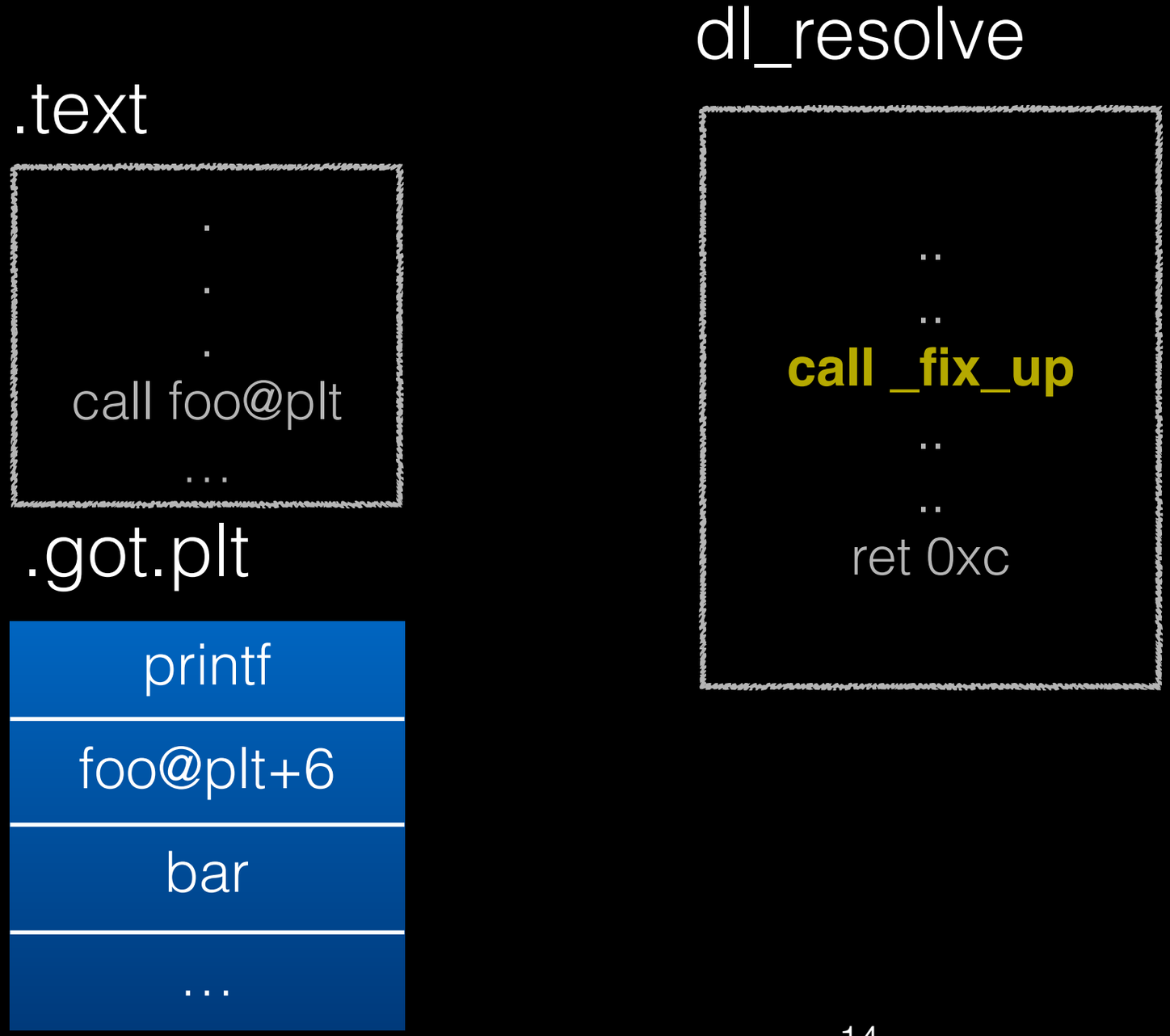
PLT0

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

jmp dl_runtime_resolve

dl_runtime_resolve (link_map,index)

Lazy binding



Lazy binding

.text

.

.

.

call foo@plt

…

.got.plt

printf

foo

bar

…

dl_resolve

..

..

..

call _fix_up

..

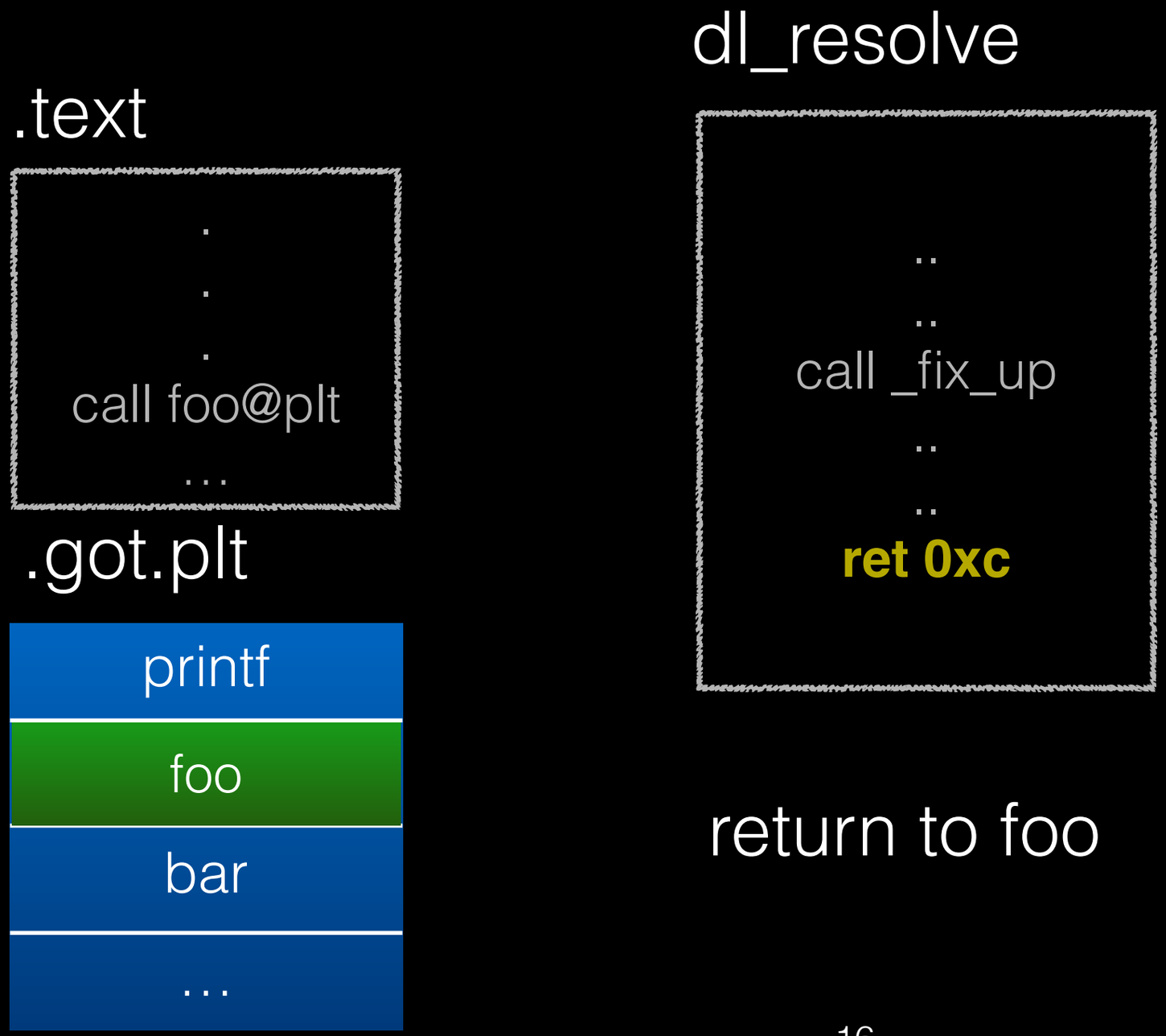
..

..

ret 0xc

找到 foo 在 library 的位置後
會填回 .got.plt

Lazy binding



Return to dl_resolve

- 利用 Lazy binding 找尋 library 中函式位置的方式，取得 system 的位置
  - 不需要 leak memory 就可以取得 system 位置
  - 必須可以控制 resolve 的參數，或是可以改到 .dynamic
- 必須在沒有 PIE 及 RELRO 的情況下才可利用
  - PIE 必須 leak code 段，相對沒有必要用
  - RELRO 則 link_map 和 dl_resolve 都會被填成 0

Return to dl_resolve

- 一些相關的 Dynamic entry
  - DT_JMPREL
  - DT_SYMTAB
  - DT_STRTAB
  - DT_VERSYM

```
angelboy@angelboy-adl:~/wargame$ readelf -d magic

Dynamic section at offset 0xf14 contains 24 entries:
  Tag        Type                         Name/Value
 0x00000001 (NEEDED)                     Shared library: [libc.so.6]
 0x0000000c (INIT)                       0x8048414
 0x0000000d (FINI)                       0x8048814
 0x00000019 (INIT_ARRAY)                 0x8049f08
 0x0000001b (INIT_ARRAYSZ)               4 (bytes)
 0x0000001a (FINI_ARRAY)                 0x8049f0c
 0x0000001c (FINI_ARRAYSZ)               4 (bytes)
 0x6ffffef5 (GNU_HASH)                   0x80481ac
 0x00000005 (STRTAB)                     0x80482c0
 0x00000006 (SYMTAB)                     0x80481d0
 0x0000000a (STRSZ)                      150 (bytes)
 0x0000000b (SYMENT)                     16 (bytes)
 0x00000015 (DEBUG)                      0x0
 0x00000003 (PLTGOT)                     0x804a000
 0x00000002 (PLTRELSZ)                   96 (bytes)
 0x00000014 (PLTREL)                     REL
 0x00000017 (JMPREL)                     0x80483b4
 0x00000011 (REL)                        0x80483a4
 0x00000012 (RELSZ)                      16 (bytes)
 0x00000013 (RELENT)                     8 (bytes)
 0x6ffffffe (VERNEED)                    0x8048374
 0x6fffffff (VERNEEDNUM)                 1
 0x6ffffff0 (VERSYM)                     0x8048356
 0x00000000 (NULL)                       0x0
```

Return to dl_resolve

- DT_JMPREL
  - address of PLT relocs
  - Tag 0x17
- PLT relocs
  - 存的 struct 為 Elf32_Rel
- r_offset
  - .got.plt 的位置
- r_info
  - symbol index + relocation type

```
496 typedef struct
497 {
498   Elf32_Addr    r_offset;
499   Elf32_Word    r_info;
500 } Elf32_Rel;
501
507 typedef struct
508 {
509   Elf64_Addr    r_offset;
510   Elf64_Xword   r_info;
511 } Elf64_Rel;
512
```

Return to dl_resolve

- JMPREL

```
gdb-peda$ x/30x 0x80483b4
0x80483b4:      0x0804a00c      0x00000107      0x0804a010      0x00000207
0x80483c4:      0x0804a014      0x00000307      0x0804a018      0x00000407
0x80483d4:      0x0804a01c      0x00000507      0x0804a020      0x00000607
0x80483e4:      0x0804a024      0x00000707      0x0804a028      0x00000807
0x80483f4:      0x0804a02c      0x00000907      0x0804a030      0x00000a07
0x8048404:      0x0804a034      0x00000b07      0x0804a038      0x00000c07
```

r_offset　　r_info

r_info 中的 0x07 為 R_386_JMP_SLOT

Return to dl_resolve

- DT_SYMTAB
  - address of symbol table
- Symbol table
  - 存的 struct 為 Elf32_Symol
- st_name
  - index of string table
- st_value (symbol value)
- st_size (symbol size)
- st_info (symbol type and binding)
- st_other (symbol visibility)
- st_shndx (section index)

```
381 typedef struct
382 {
383   Elf32_Word    st_name;
384   Elf32_Addr    st_value;
385   Elf32_Word    st_size;
386   unsigned char st_info;
387   unsigned char st_other;
388   Elf32_Section st_shndx;
389 } Elf32_Sym;

391 typedef struct
392 {
393   Elf64_Word    st_name;
394   unsigned char st_info;
395   unsigned char st_other;
396   Elf64_Section st_shndx;
397   Elf64_Addr    st_value;
398   Elf64_Xword   st_size;
399 } Elf64_Sym;
```

Return to dl_resolve

- SYMTAB

```
gdb-peda$ x/32x 0x80481d0
0x80481d0:    0x00000000    0x00000000    0x00000000    0x00000000
0x80481e0:    0x0000004e    0x00000000    0x00000000    0x00000012
0x80481f0:    0x00000040    0x00000000    0x00000000    0x00000012
0x8048200:    0x0000001a    0x00000000    0x00000000    0x00000012
0x8048210:    0x0000003b    0x00000000    0x00000000    0x00000012
0x8048220:    0x00000036    0x00000000    0x00000000    0x00000012
0x8048230:    0x0000005a    0x00000000    0x00000000    0x00000012
0x8048240:    0x00000073    0x00000000    0x00000000    0x00000020
```

st_name

st_other

Return to dl_resolve

- DT_STRTAB
  - address of string table

```
gdb-peda$ x/20s 0x80482c0
0x80482c0:      ""
0x80482c1:      "libc.so.6"
0x80482cb:      "_IO_stdin_used"
0x80482da:      "fflush"
0x80482e1:      "srand"
0x80482e7:      "__isoc99_scanf"
0x80482f6:      "puts"
0x80482fb:      "time"
0x8048300:      "printf"
0x8048307:      "strlen"
0x804830e:      "read"
0x8048313:      "stdout"
0x804831a:      "system"
0x8048321:      "__libc_start_main"
0x8048333:      "__gmon_start__"
0x8048342:      "GLIBC_2.7"
0x804834c:      "GLIBC_2.0"
0x8048356:      ""
0x8048357:      ""
```

dl_resolve in x86

Return to dl_resolve

- dl_resolve(link_map,index)
  - Elf32_Rel *reloc = JMPREL + index
  - Elf32_Sym *sym = &SYMTAB[((reloc->r_info)>>8)]
  - assert (((reloc->r_info)&0xff) == 0x7)
    - type 檢查，檢查是否為 R_386_JMP_SLOT

Return to dl_resolve

- dl_resolve(link_map,index)
  - if ((ELFW(ST_VISIBILITY) (sym->st_other), 0) == 0)
  - => if (sym->st_other) & 3 == 0)
  - 判斷該 symbol 是否已經解析過
  - 如果解析過則不去 lookup

Return to dl_resolve

- uint16_t ndx = VERSYM[(reloc->r_info) >> 8]
  - r_found_version *version = &l->l_version[ndx]
    - 決定 symbol version，ndx 如果是 0 則使用 local symbol
- name = STRTAB + sym->st_name
- 最後會利用這個 name 去找尋該 function 在 library 中的位置

dl_resolve in x64

Return to dl_resolve

- dl_resolve(link_map,index) - x64
  - Elf64_Rel *reloc = JMPREL + index*3*8
  - Elf64_Sym *sym = &SYMTAB[((reloc->r_info)>>0x20)]
    - i.e. *sym = DT_SYMTAB + (reloc->r_info)*3*8
- assert (((reloc->r_info)&0xff) == 0x7)
  - type 檢查，檢查是否為 R_X86_64_JMP_SLOT

Return to dl_resolve

- dl_resolve(link_map,index)
  - if ((ELFW(ST_VISIBILITY) (sym->st_other), 0) == 0)
  - => if (sym->st_other) & 3 == 0)
  - 判斷該 symbol 是否已經解析過
  - 如果解析過則不去 lookup

Return to dl_resolve

- uint16_t ndx = VERSYM[(reloc->r_info) >> 0x20]
  - r_found_version *version = &l->l_version[ndx]
  - 決定 symbol version，ndx 如果是 0 則使用 local symbol
    - 但在偽造 sym 之後 VERSYM 取值會超出範圍，造成 segment fault ，解決方式是讓 link_map + 0x1c8 的地方變為 0 ，不過相對要 leak link_map address
- name = STRTAB + sym->st_name
- 最後會利用這個 name 去找尋該 function 在 library 中的位置

Return to dl_resolve

- 利用方式
- 控制 eip 到 dl_resolve
  - 需給定 link_map 及 index 兩個參數
    - 可只接利用 PLT0 的程式碼，這樣只要傳一個index 的參數就好
- 控制 index 大小，將 reloc 的位置控制在自己的掌控範圍內
- 偽造 reloc 的內容，使得 sym 也落在自己掌控範圍內
- 偽造 sym 的內容，使得最後 name 的位置也在自己掌控範圍內
- name = system and return to system

Return to dl_resolve

- 需特別注意的地方
  - uint16_t ndx = VERSYM[(reloc->r_info) >> 8]
  - 最好使得 ndx 的結果是 0，不然可能會找不到
- 偽造 reloc 時
  - r_offset 必須是可寫的，原因是為了他要將找到的 function 寫回 .got.plt
  - 可以利用這點寫去別的 function 的 .got.plt 達到 GOT Hijacking

Codegate Final

yocto

程式行為及漏洞

- 將使用者的 input 讀入之後，shutdown 0,1,2
- 使用者的 input 會存在固定位置的 .bss 段
- 程式會將使用者的 input 以 “.” 分開，並判斷是否為整數，當輸入為 1111.2222.3333 時，可將 eip 控制到 3333 而 2222 , 1111 分別為 arg1,arg2

漏洞利用

- 利用 return to dl_resolve (PLT0) 改寫 atoi 的 got
  - aaaa.bbbb.cccc.dddd
    - aaaa 為從 dl_resolve 後 return 回來的位置，因此可二次控制 EIP ，將它控制在 atoi 附近，使得 atoi 讀入 dddd 便可達成 system(dddd)
- 不過因為 input 被關起來了，必須做 reverse shell 或是直接 cat flag

Other dl_resolve

Return to dl_resolve

- 偽造 link_map 結構，並故意用解析過 symbol 的條件，進行跳轉
  - 需求
    - 已知 libc 版本
    - 須偽造 link_map 、elf_rel、elf_sym
    - 有一個已經 resolve 過的 function

Return to dl_resolve

- 先偽造 link_map、index
  - 令 link_map 指向 got.plt 上以解析過的 function
    - link_map 中的 JMPREL、SYMTAB 及 STRTAB 要正確或是可以去 dereference 否則會 segfault ，也會影響到之後的 REL 跟 SYM
- 偽造 index
- retun to dl_resolve
- 偽造 Elf64_Rel *reloc = JMPREL + index*3*8
- 偽造 Elf64_Sym *sym = &SYMTAB[((reloc->r_info)>>0x20)]

Return to dl_resolve

- if((sym->st_other) & 3 != 0) //如果已 resolve 過
  - value = DL_FIXUP_MAKE_VALUE (l, l->l_addr + sym->st_value);
- l->l_addr 則是剛剛我們所偽造的 link_map 指向位置
- sym->st_value 在偽造時先放好已 reslove 過的 function 與 system 之間差 offset （有可能是負）
- 最後就會 resolve 出 system
- 須注意最後會寫到會把結果也就是 system 寫到 reloc->offset + system 的地方，要讓他是可以寫的

Reference

- x64でROP stager + Return-to-dl-resolveによるASLR+DEP回避をやってみる
- Advanced Return to libc Exploits
- glibc_dl-runtime.c