

Execution

Angelboy @ bamboofox

Who am I

- Angelboy
- 中央大學碩士班 ADLab
- Bamboofox / HITCON 成員
- 擅長領域：PWN



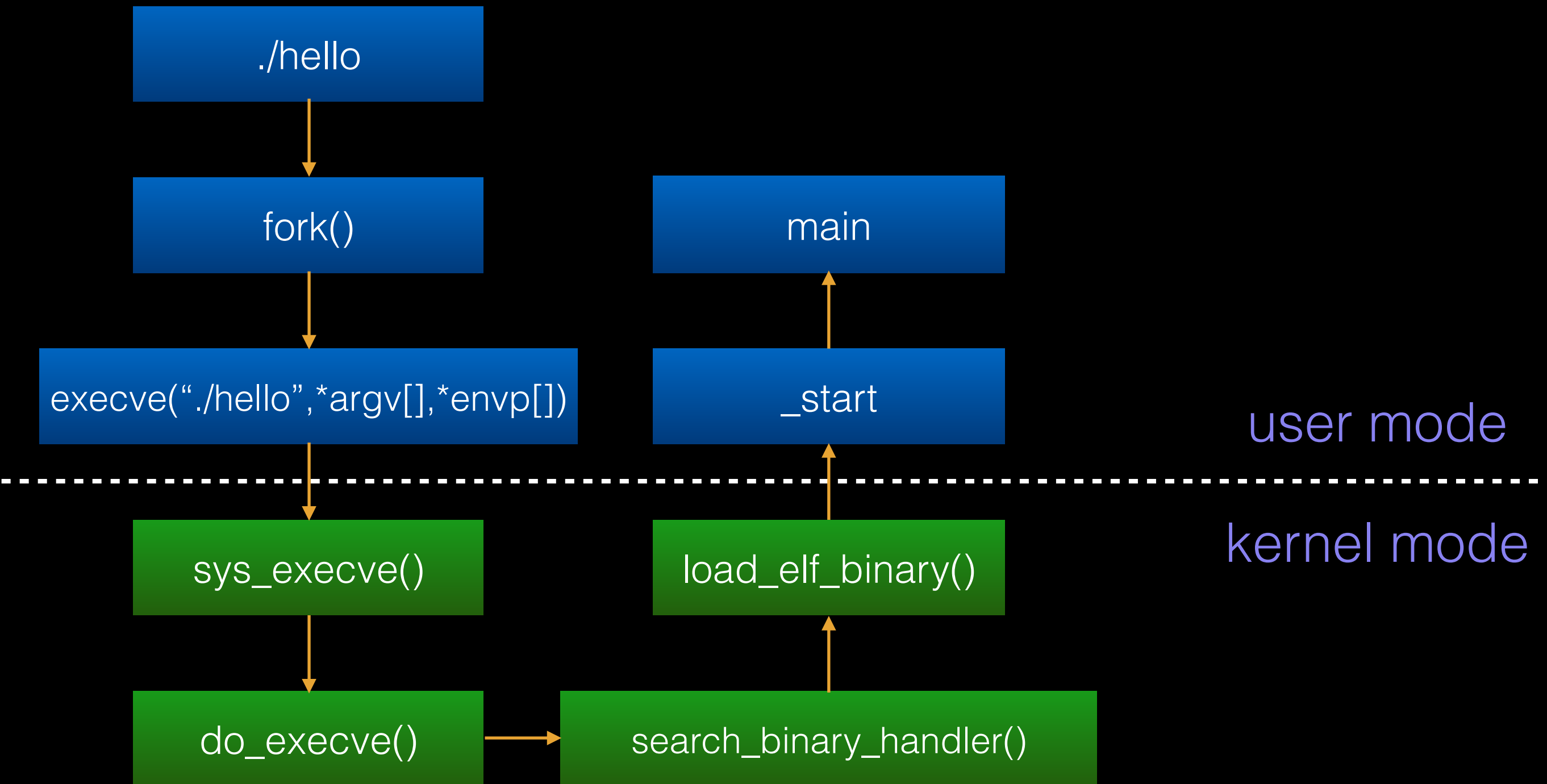
Outline

- How programs get run
- Lazy binding
 - Global Offset Table
- GOT Hijacking

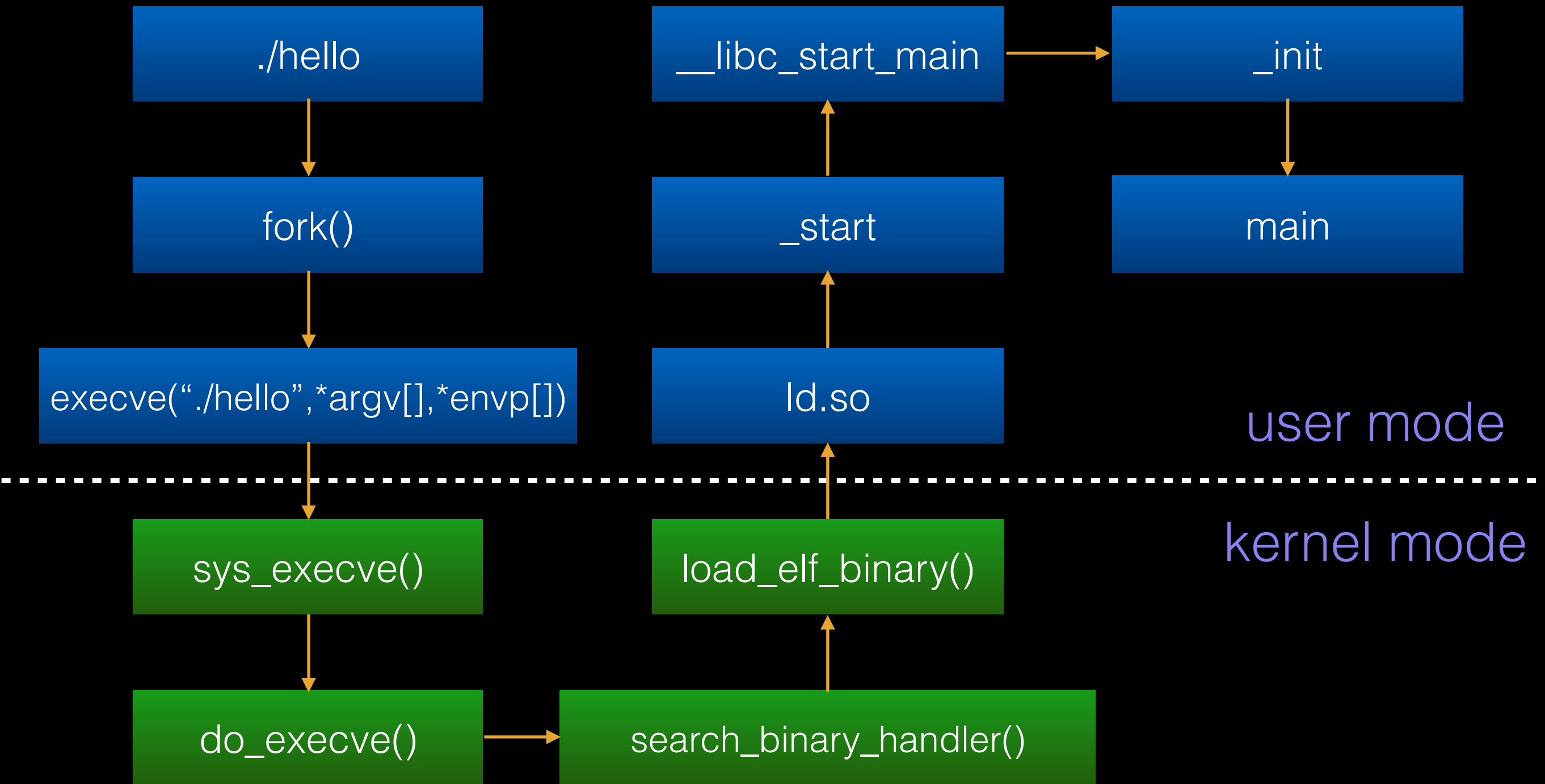
How programs get run

- We focus on ELF (Executable and Linking Format)
- What's happened when we execute an elf file.
 - ex : ./hello

Overview of the workflow (static linking)



Overview of the workflow (dynamic linking)



How programs get run

- `sys_execve()`
 - 檢查參數 ex: argv , envp
- `do_execve()`
 - 搜尋執行檔位置
 - 讀取執行檔前 128 byte 獲取執行檔的資訊
 - ex: magic number

How programs get run

- `search_binary_handler()`
 - 利用前面所獲取的資訊來呼叫相對應的 handler
 - ex : `load_script()` 、 `load_elf_binary()`

How programs get run

- `load_elf_binary()`
 - 檢查及獲取 program header 資訊
 - 如果是 dynamic linking 則利用 .interp 這個 section 來確定 loader 路徑
 - 將 program header 紀錄的位置 mapping 到 memory 中，ex : code segment 位置
 - 將 `sys_execve` 的 return address 改為 loader (ld.so) 的 entry point
 - static linking 下則會是 elf 的 entry point

How programs get run

- How program maps to virtual memory.
- 在 program header 中
 - 記錄著哪些 segment 應該 mapping 到什麼位置，以及該 segment 的讀寫執行權限
 - 記錄哪些 section 屬於哪些 segment
 - 當 program mapping 記憶體時會根據權限的不同來分成好幾個 segment

How programs get run

- How program maps to virtual memory.

```
angelboy@angelboy-adl:~$ readelf -l hello
```

```
Elf file type is EXEC (Executable file)
```

```
Entry point 0x8048350
```

```
There are 9 program headers, starting at offset 52
```

```
Program Headers:
```

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
PHDR	0x000034	0x08048034	0x08048034	0x00120	0x00120	R E	0x4
INTERP	0x000154	0x08048154	0x08048154	0x00013	0x00013	R	0x1
[Requesting program interpreter: /lib/ld-linux.so.2]							
LOAD	0x000000	0x08048000	0x08048000	0x005f8	0x005f8	R E	0x1000
LOAD	0x000f08	0x08049f08	0x08049f08	0x0011c	0x00120	RW	0x1000
DYNAMIC	0x000f14	0x08049f14	0x08049f14	0x000e8	0x000e8	RW	0x4
NOTE	0x000168	0x08048168	0x08048168	0x00044	0x00044	R	0x4
GNU_EH_FRAME	0x00051c	0x0804851c	0x0804851c	0x0002c	0x0002c	R	0x4
GNU_STACK	0x000000	0x00000000	0x00000000	0x00000	0x00000	RW	0x10
GNU_RELRO	0x000f08	0x08049f08	0x08049f08	0x000f8	0x000f8	R	0x1

```
Section to Segment mapping:
```

Segment	Sections
00	
01	.interp
02	.interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rel.dyn .rel.plt .init .plt .text .fini .rodata .eh_frame_hdr .eh_frame
03	.init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
04	.dynamic
05	.note.ABI-tag .note.gnu.build-id
06	.eh_frame_hdr
07	
08	.init_array .fini_array .jcr .dynamic .got

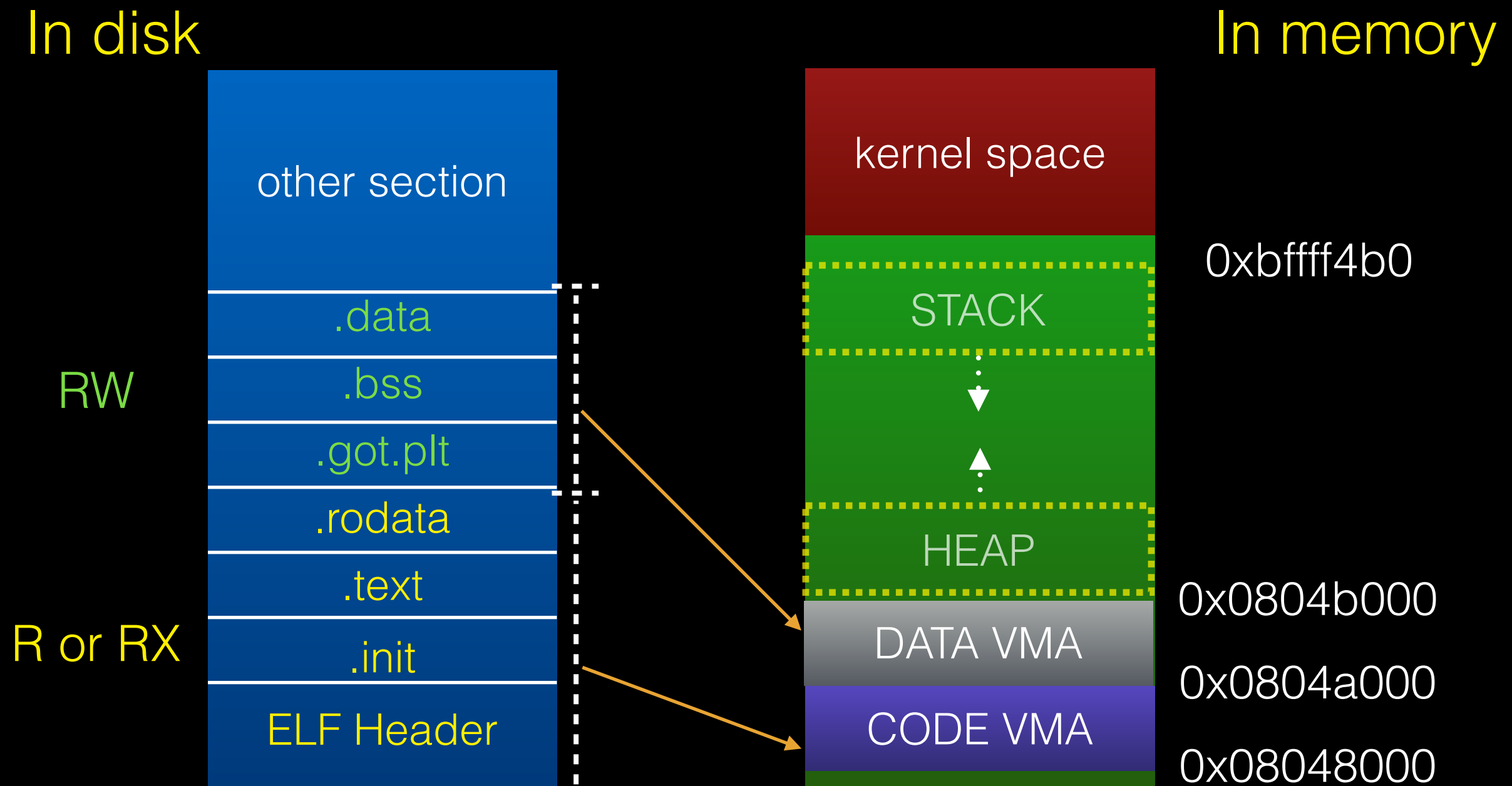
權限

mapping 位置

segment 中有哪些 section

How programs get run

- How program maps to virtual memory.



How programs get run

- ld.so
 - 載入 elf 所需的 shared library
 - 這部分會記錄在 elf 中的 DT_NEED 中
 - 初始化 GOT
 - 其他相關初始化的動作
 - ex : 將 symbol table 合併到 global symbol table
等等

How programs get run

- ld.so 其實是個很複雜的程式，要認真全部講完，一兩天的時間絕對不夠用
- 有興趣可參考 [elf/rtld.c](#)
- 這邊最重要的是在執行程式時，要知道不是從 main 開始的

How programs get run

- `_start`
 - 將下列項目傳給 `libc_start_main`
 - 環境變數起始位置
 - `.init`
 - 呼叫 `main` 之前的初始化工作
 - `.fini`
 - 程式結束前的收尾工作

How programs get run

- `_libc_start_main`

- 執行 `.init`
- 執行 `main`
- 執行 `.fini`
- 執行 `exit`

Lazy binding

- Dynamic linking 的程式在執行過程中，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

Global Offset Table

- library 的位置再載入後才決定，因此無法在 compile 後，就知道 library 中的 function 在哪，該跳去哪
- GOT 為一個函式指標陣列，儲存其他 library 中，function 的位置，但因 Lazy binding 的機制，並不會一開始就把正確的位置填上，而是填上一段 plt 位置的 code

Global Offset Table

- 當執行到 library 的 function 時才會真正去尋找 function ，最後再把 GOT 中的位置填上真正 function 的位置

```
80484e8:  c7 04 24 b0 85 04 08    mov     DWORD PTR [esp],0x80485b0
80484ef:  e8 7c fe ff ff          call    8048370 <puts@plt>
80484f4:  b8 ff ff ff ff          mov     eax,0xffffffff
```

Global Offset Table

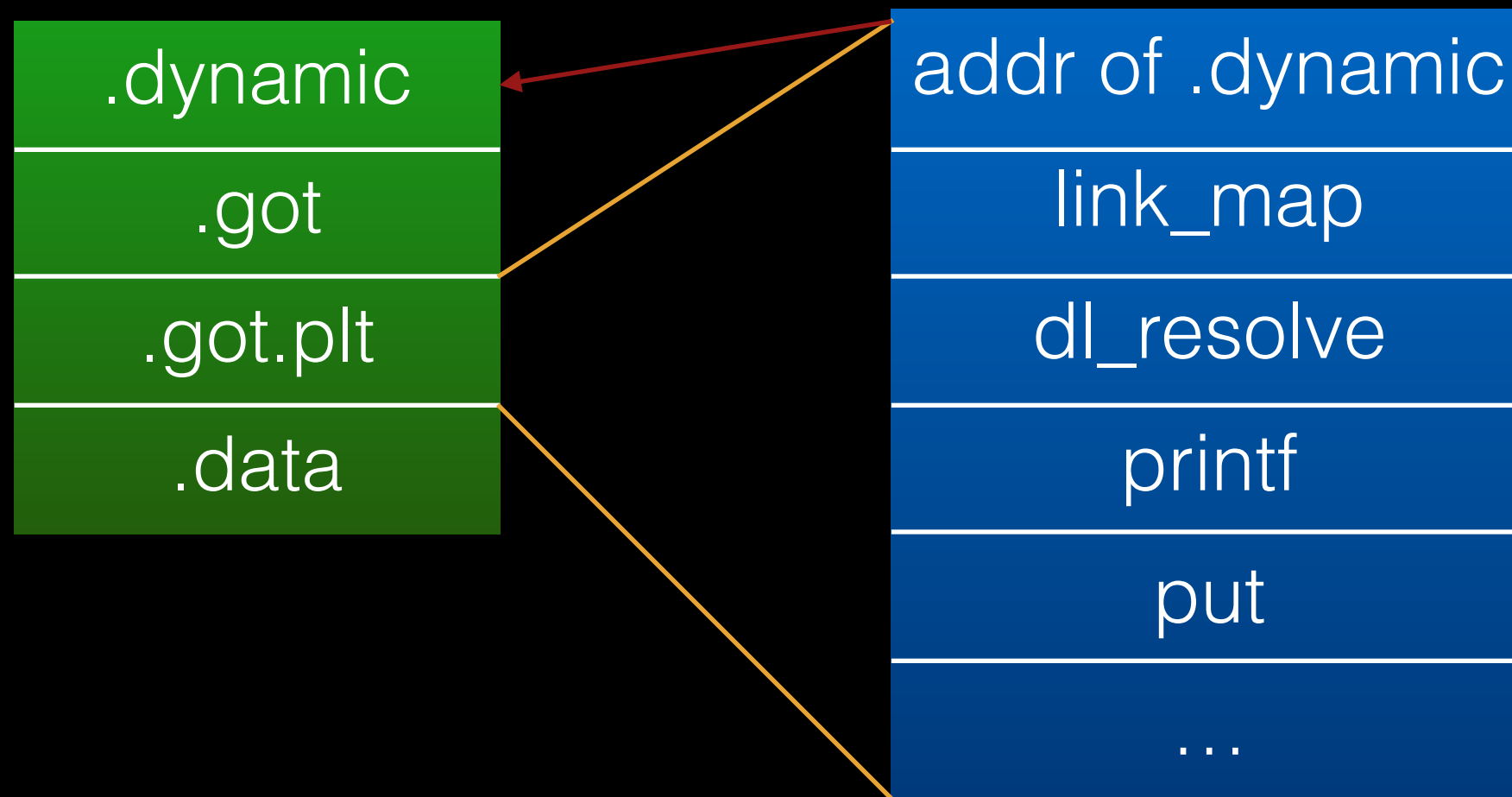
- 分成兩部分
 - .got
 - 保存全域變數引用位置
 - .got.plt
 - 保存函式引用位置

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`
 - 用來找出函式位置的函式
 - 後面則是程式中 `.so` 函式引用位置

Global Offset Table

- layout



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@plt+6
bar
...



Lazy binding

.text

```
·  
·  
·  
call foo@plt  
...
```

.got.plt

printf
foo@plt+6
bar
...

foo@plt

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

PLT0

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


Lazy binding

.text

```
·  
·  
·  
call foo@plt  
...
```

.got.plt

printf
foo@plt+6
bar
...

foo@plt

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

PLT0

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

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

.text

```
·  
·  
·  
call foo@plt  
...
```

foo@plt

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

PLT0

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

.got.plt

printf
foo@plt+6
bar
...



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@plt+6
bar
...



Lazy binding

.text

```
·  
·  
·  
call foo@plt  
...
```

.got.plt

printf
foo@plt+6
bar
...

foo@plt

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

push link_map

PLT0

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

Lazy binding

.text

```
·  
·  
·  
call foo@plt  
...
```

.got.plt

printf
foo@plt+6
bar
...

foo@plt

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

PLT0

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

jmp dl_runtime_resolve

→ dl_runtime_resolve(link_map, index)

Lazy binding

.text

```
    .  
    .  
    .  
    call foo@plt  
    ...
```

.got.plt

printf
foo@plt+6
bar
...

dl_resolve

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

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

.text

```
    .  
    .  
    .  
    call foo@plt  
    ...
```

.got.plt

printf
foo
bar
...

dl_resolve

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

return to foo

Lazy binding

- 第二次 call foo 時

.text

```
·  
·  
·  
call foo@plt  
...
```

.got.plt

printf
foo
bar
...

foo@plt

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

PLT0

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

Lazy binding

- 第二次 call foo 時

.text

```
·  
·  
·  
call foo@plt  
...
```

.got.plt

printf
foo
bar
...

foo@plt

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

PLT0

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

Jmp to foo function

How to find the GOT

- `objdump -R elf` or `readelf -r elf`

```
angelboy@angelboy-adl:~$ objdump -R hello
```

```
hello:      file format elf32-i386
```

DYNAMIC RELOCATION RECORDS

OFFSET	TYPE	VALUE
08049ffc	R_386_GLOB_DAT	__gmon_start__
0804a00c	R_386_JUMP_SLOT	__stack_chk_fail@GLIBC_2.4
0804a010	R_386_JUMP_SLOT	strcpy@GLIBC_2.0
0804a014	R_386_JUMP_SLOT	puts@GLIBC_2.0
0804a018	R_386_JUMP_SLOT	__gmon_start__
0804a01c	R_386_JUMP_SLOT	__libc_start_main@GLIBC_2.0

GOT Hijacking

- 為了實作 *Lazy binding* 的機制 GOT 位置必須是可寫入的
- 但如果程式有存在任意更改位置的漏洞，便可改寫 GOT，造成程式流程的改變
- 也就是控制 eip

GOT Hijacking

- 第二次 call foo 時

.text

```
·  
·  
·  
call foo@plt  
...
```

.got.plt

printf
foo
bar
...

foo@plt

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

PLT0

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

GOT Hijacking

- 第二次 call foo 時

.text

```
·  
·  
·  
call foo@plt  
...
```

.got.plt

printf
system
bar
...

foo@plt

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

PLT0

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

GOT Hijacking

- 第二次 call foo 時

.text

```
·  
·  
·  
call foo@plt  
...
```

.got.plt

printf
system
bar
...

foo@plt

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

PLT0

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

Jmp to **system** function

Reference

- Glibc cross reference
- Linux Cross Reference
- 程式設計師的自我修養

Q & A