Linux Binary Exploitation

Basic Knowledge x86-64

angelboy@chroot.org

Outline

- Introduction
- Section
- Compilation Flow
- Execution
- x86 assembbly

Outline

- Introduction
- Section
- Compilation Flow
- Execution
- x86 assembbly

Introduction

- Reverse Engineering
- Exploitation
- Useful Tool

Reverse Engineering

- 正常情況下我們不容易取得執行檔的原始碼，所以我們很常需逆向分析程式尋找漏洞
- Static Analysis
- Dynamic Analysis

Reverse Engineering

- Static Analysis
  - Analyze program without running
  - e.g.
    - *objdump*
      - Machine code to asm

```
00000000000013af <main>:
    13af:       55                      push   rbp
    13b0:       48 89 e5                mov    rbp,rsp
    13b3:       48 83 ec 10             sub    rsp,0x10
    13b7:       b8 00 00 00 00          mov    eax,0x0
    13bc:       e8 57 fe ff ff          call   1218 <init_proc>
    13c1:       b8 00 00 00 00          mov    eax,0x0
    13c6:       e8 6c ff ff ff          call   1337 <menu>
    13cb:       b8 00 00 00 00          mov    eax,0x0
    13d0:       e8 90 f8 ff ff          call   c65 <read_int>
    13d5:       89 45 fc                mov    DWORD PTR [rbp-0x4],eax
    13d8:       8b 45 fc                mov    eax,DWORD PTR [rbp-0x4]
    13db:       83 f8 02                cmp    eax,0x2
    13de:       74 24                   je     1404 <main+0x55>
    13e0:       83 f8 02                cmp    eax,0x2
    13e3:       7f 07                   jg     13ec <main+0x3d>
    13e5:       83 f8 01                cmp    eax,0x1
    13e8:       74 0e                   je     13f8 <main+0x49>
    13ea:       eb 46                   jmp    1432 <main+0x83>
    13ec:       83 f8 03                cmp    eax,0x3
    13ef:       74 1f                   je     1410 <main+0x61>
    13f1:       83 f8 04                cmp    eax,0x4
    13f4:       74 26                   je     141c <main+0x6d>
    13f6:       eb 3a                   jmp    1432 <main+0x83>
    13f8:       b8 00 00 00 00          mov    eax,0x0
    13fd:       e8 35 f9 ff ff          call   d37 <build>
    1402:       eb 3a                   jmp    143e <main+0x8f>
    1404:       b8 00 00 00 00          mov    eax,0x0
    1409:       e8 d8 fa ff ff          call   ee6 <see>
    140e:       eb 2e                   jmp    143e <main+0x8f>
    1410:       b8 00 00 00 00          mov    eax,0x0
    1415:       e8 62 fc ff ff          call   107c <upgrade>
    141a:       eb 22                   jmp    143e <main+0x8f>
```

Reverse Engineering

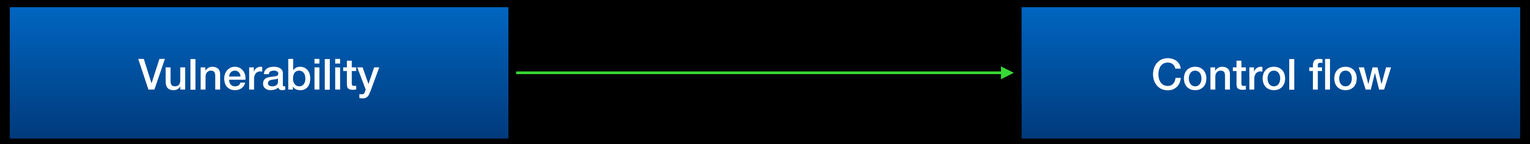
- Dynamic Analysis
  - Analyze program with running
  - e.g.
    - *strace*
      - trace all system call
    - *ltrace*
      - trace all library call

```
angelboy@ubuntu:~$ ltrace id
__libc_start_main(0x401ac0, 1, 0x7ffcf6fdd668, 0x406150 <unfinished ...>
is_selinux_enabled(1, 0x7ffcf6fdd668, 0x7ffcf6fdd678, 0)
strrchr("id", '/')
setlocale(LC_ALL, "")
bindtextdomain("coreutils", "/usr/share/locale")
textdomain("coreutils")
__cxa_atexit(0x402cf0, 0, 0, 0)
getopt_long(1, 0x7ffcf6fdd668, "agnruzGZ", 0x406a00, nil)
getenv("POSIXLY_CORRECT")
__errno_location()
geteuid()
__errno_location()
getuid()
__errno_location()
getegid()
getgid()
dcgettext(0, 0x4063ab, 5, 0x609340)
__printf_chk(1, 0x4063ab, 0x609340, 0)
getpwuid(1000, 8, 0x7f5022dc1780, 0x7ffffff7)
__printf_chk(1, 0x40639c, 0x1f19860, 0x7ffcf6fdd4a0)
dcgettext(0, 0x4063a1, 5, 0x609320)
__printf_chk(1, 0x4063a1, 0x609320, 0)
getgrgid(1000, 9, 0x7f5022dc1780, 0x7ffffff6)
__printf_chk(1, 0x40639c, 0x1f1cb60, 0x7ffcf6fdd4a0)
getgroups(0, 0, 0x7ffcf6fdd540, 0x7ffffff6)
```

Exploitation

- 利用漏洞來達成攻擊者目的
- 一般來說主要目的在於取得程式控制權
- 又稱 Pwn

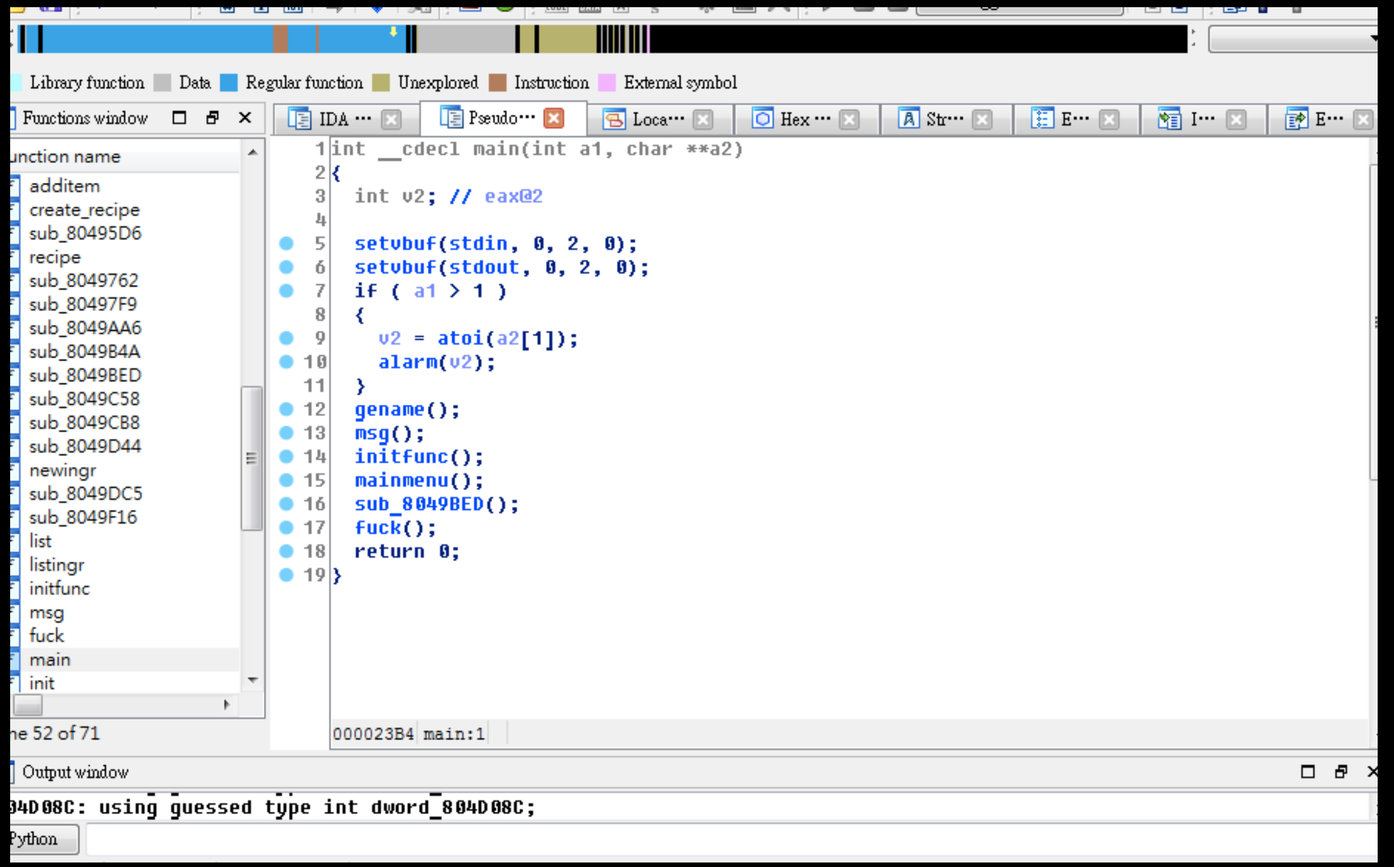
Exploitation



- Binary exploitation
  - 專指與 binary 相關的漏洞利用
  - 本課程重點

Useful Tool

- IDA PRO - a static analysis tool



Useful Tool

- GDB - a dynamic analysis tool
  - The GNU Project Debugger

```
(gdb) disas main
Dump of assembler code for function main:
   0x0000000000400626 <+0>:     push   %rbp
   0x0000000000400627 <+1>:     mov    %rsp,%rbp
   0x000000000040062a <+4>:     sub    $0x30,%rsp
   0x000000000040062e <+8>:     mov    %fs:0x28,%rax
   0x0000000000400637 <+17>:    mov    %rax,-0x8(%rbp)
   0x000000000040063b <+21>:    xor    %eax,%eax
   0x000000000040063d <+23>:    mov    $0x400724,%esi
   0x0000000000400642 <+28>:    mov    $0x400726,%edi
=> 0x0000000000400647 <+33>:    callq  0x400510 <fopen@plt>
   0x000000000040064c <+38>:    mov    %rax,-0x28(%rbp)
   0x0000000000400650 <+42>:    mov    -0x28(%rbp),%rdx
   0x0000000000400654 <+46>:    lea    -0x20(%rbp),%rax
   0x0000000000400658 <+50>:    mov    %rdx,%rcx
   0x000000000040065b <+53>:    mov    $0x1,%edx
   0x0000000000400660 <+58>:    mov    $0x14,%esi
   0x0000000000400665 <+63>:    mov    %rax,%rdi
   0x0000000000400668 <+66>:    callq  0x4004e0 <fread@plt>
   0x000000000040066d <+71>:    lea    -0x20(%rbp),%rax
   0x0000000000400671 <+75>:    mov    %rax,%rdi
   0x0000000000400674 <+78>:    callq  0x4004d0 <puts@plt>
   0x0000000000400679 <+83>:    mov    $0x0,%eax
   0x000000000040067e <+88>:    mov    -0x8(%rbp),%rcx
   0x0000000000400682 <+92>:    xor    %fs:0x28,%rcx
   0x000000000040068b <+101>:   je     0x400692 <main+108>
   0x000000000040068d <+103>:   callq  0x4004f0 <__stack_chk_fail@plt>
   0x0000000000400692 <+108>:   leaveq
   0x0000000000400693 <+109>:   retq
End of assembler dump.
(gdb)
```

Useful Tool

- Basic command
  - run - 執行
  - disas function name - 反組譯某個 function
  - break *0x400566 - 設斷點
  - info breakpoint - 查看已設定哪些中斷點
  - info register 查看所有 register 狀態

Useful Tool

- Basic command
  - x/wx address - 查看 address 中的內容
    - w 可換成 b/h/g 分別是取 1/2/8 byte
    - / 後可接數字 表示一次列出幾個
    - 第二個 x 可換成 u/d/s/i 以不同方式表示
      - u : unsigned int
      - d : 10 進位
      - s : 字串
      - i : 指令

Useful Tool

- Basic command
  - x/gx address 查看 address 中的内容
    - e.g.

```
gdb-peda$ x/gx 0x601030
0x601030:       0x0000000000400506
```

Useful Tool

- Basic command
  - ni - next instruction
  - si - step into
  - backtrace - 顯示上層所有 stack frame 的資訊
  - continue

Useful Tool

- Basic command
  - set *address=value
    - 將 address 中的值設成 value 一次設 4 byte
    - 可將 * 換成 {char/short/long} 分別設定 1/2/8 byte
    - e.g.
      - set *0x602040=0xdeadbeef
      - set {int}0x602040=1337

Useful Tool

- Basic command
  - 在有 debug symbol 下
    - list : 列出 source code
    - b 可直接接行號斷點
    - info local : 列出區域變數
    - print val : 印出變數 val 的值

Useful Tool

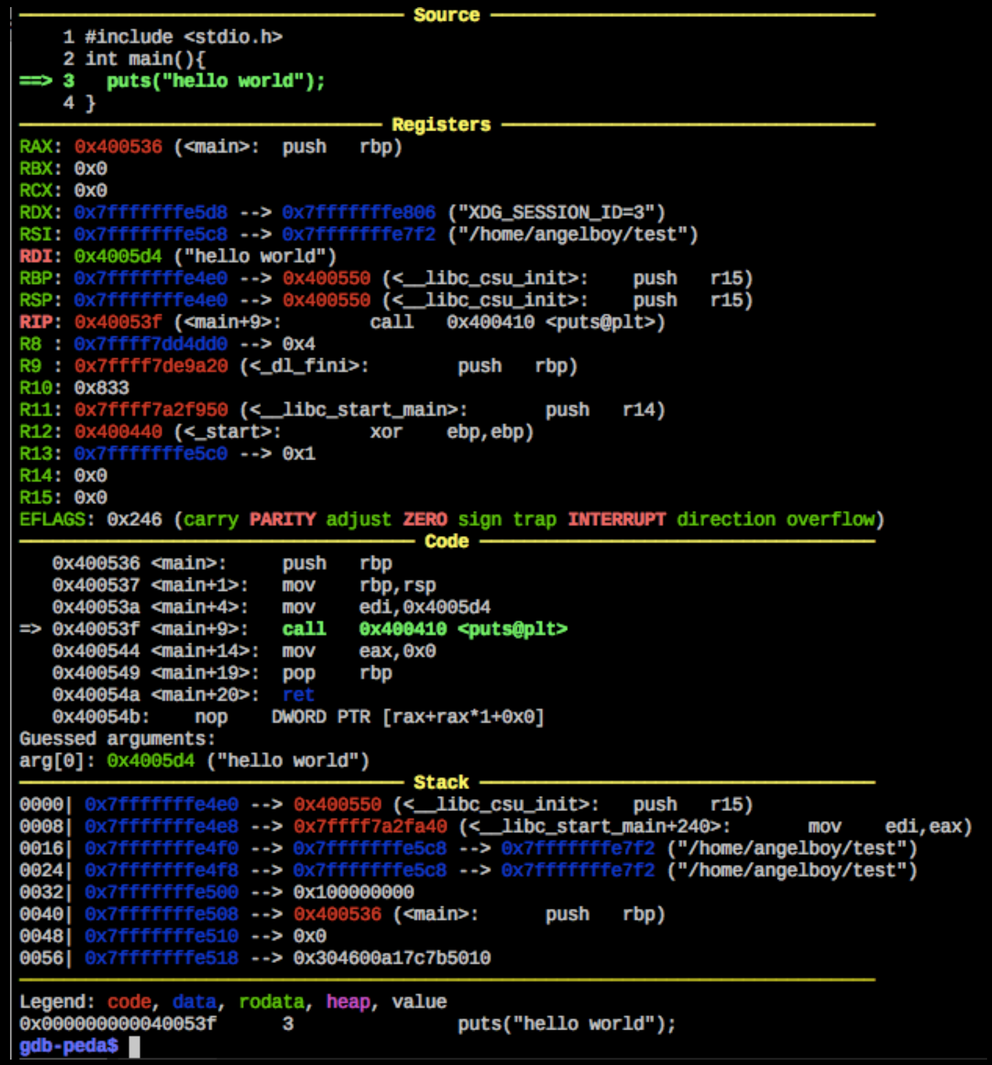
- Basic command
  - attach pid : attach 一個正在運行的 process
    - 可以配合 ncat 進行 exploit 的 debug
      - ncat -ve ./a.out -kl 8888
    - echo 0 > /proc/sys/kernel/yama/ptrace_scope

Useful Tool

- GDB - PEDA
  - Python Exploit Development Assistance for GDB
  - https://github.com/longld/peda
  - https://github.com/scwuaptx/peda

GDB - PEDA

- Screenshot



GDB - PEDA

- Some useful feature
  - checksec : Check for various security options of binary
  - elfsymbol : show elf .plt section
  - vmmap : show memory mapping
  - readelf : Get headers information from an ELF file
  - find/searchmem : Search for a pattern in memory
  - record : record every instruction at runtime

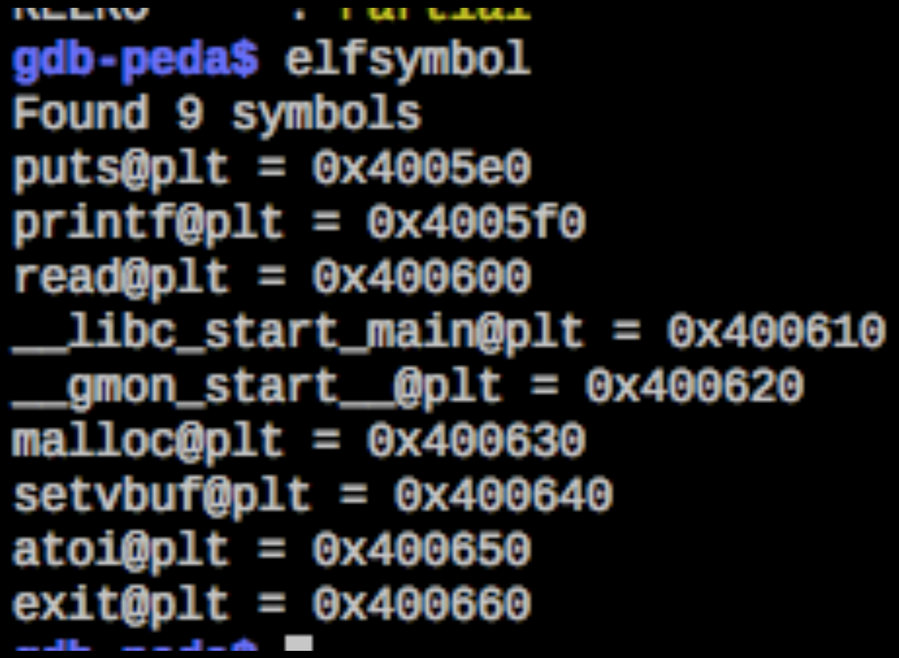
GDB - PEDA

- checksec
  - 查看 binary 中有哪些保護機制

```
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : Partial
```

GDB - PEDA

- elfsymbol
  - 查看 function .plt 做 ROP 時非常需要



GDB - PEDA

- vmmap
  - 查看 process mapping
    - 可觀察到每個 address 中的權限

```
gdb-peda$ vmmap
Start              End                Perm      Name
0x00400000         0x00401000         r-xp      /home/angelboy/ds/test
0x00600000         0x00601000         r--p      /home/angelboy/ds/test
0x00601000         0x00602000         rw-p      /home/angelboy/ds/test
0x00007ffff7a0f000 0x00007ffff7bcf000 r-xp      /lib/x86_64-linux-gnu/libc-2.21.so
0x00007ffff7bcf000 0x00007ffff7dcf000 ---p      /lib/x86_64-linux-gnu/libc-2.21.so
0x00007ffff7dcf000 0x00007ffff7dd3000 r--p      /lib/x86_64-linux-gnu/libc-2.21.so
0x00007ffff7dd3000 0x00007ffff7dd5000 rw-p      /lib/x86_64-linux-gnu/libc-2.21.so
0x00007ffff7dd5000 0x00007ffff7dd9000 rw-p      mapped
0x00007ffff7dd9000 0x00007ffff7dfd000 r-xp      /lib/x86_64-linux-gnu/ld-2.21.so
0x00007ffff7fd0000 0x00007ffff7fd3000 rw-p      mapped
0x00007ffff7ff6000 0x00007ffff7ff8000 rw-p      mapped
0x00007ffff7ff8000 0x00007ffff7ffa000 r--p      [vvar]
0x00007ffff7ffa000 0x00007ffff7ffc000 r-xp      [vdso]
0x00007ffff7ffc000 0x00007ffff7ffd000 r--p      /lib/x86_64-linux-gnu/ld-2.21.so
0x00007ffff7ffd000 0x00007ffff7ffe000 rw-p      /lib/x86_64-linux-gnu/ld-2.21.so
0x00007ffff7ffe000 0x00007ffff7fff000 rw-p      mapped
0x00007fffffffde000 0x00007ffffffff000 rw-p     [stack]
0xffffffffff600000 0xffffffffff601000 r-xp      [vsyscall]
```

GDB - PEDA

- readelf
  - 查看 section 位置
    - 有些攻擊手法會需要
    - e.g. ret2dl_resolve

```
gdb-peda$ readelf
.interp = 0x400238
.note.ABI-tag = 0x400254
.note.gnu.build-id = 0x400274
.gnu.hash = 0x400298
.dynsym = 0x4002c0
.dynstr = 0x4003e0
.gnu.version = 0x400450
.gnu.version_r = 0x400468
.rela.dyn = 0x400488
.rela.plt = 0x4004d0
.init = 0x4005a8
.plt = 0x4005d0
.text = 0x400670
.fini = 0x400904
.rodata = 0x400910
.eh_frame_hdr = 0x40091c
.eh_frame = 0x400958
.init_array = 0x600e10
.fini_array = 0x600e18
.jcr = 0x600e20
.dynamic = 0x600e28
.got = 0x600ff8
.got.plt = 0x601000
.data = 0x601060
.bss = 0x601070
```

GDB - PEDA

- find (alias searchmem)
  - search memory 中的 patten
    - 通常拿來找字串
    - e.g. /bin/sh

```
gdb-peda$ find /bin/sh
Searching for '/bin/sh' in: None ranges
Found 1 results, display max 1 items:
libc : 0x7ffff7b9b39d --> 0x68732f6e69622f ('/bin/sh')
```

GDB - PEDA

- record
  - 記錄每個 instruction 讓 gdb 可回朔前面的指令，在 PC 被改變後，可利用該功能，追回原本發生問題的地方

Useful Tool

- Pwntools
  - Exploit development library
  - python

```
from pwn import *
context(arch = 'i386', os = 'linux')

r = remote('exploitme.example.com', 31337)
# EXPLOIT CODE GOES HERE
r.send(asm(shellcraft.sh()))
r.interactive()
```

Outline

- Introduction
- Section
- Compilation Flow
- Execution
- x86 assembbly

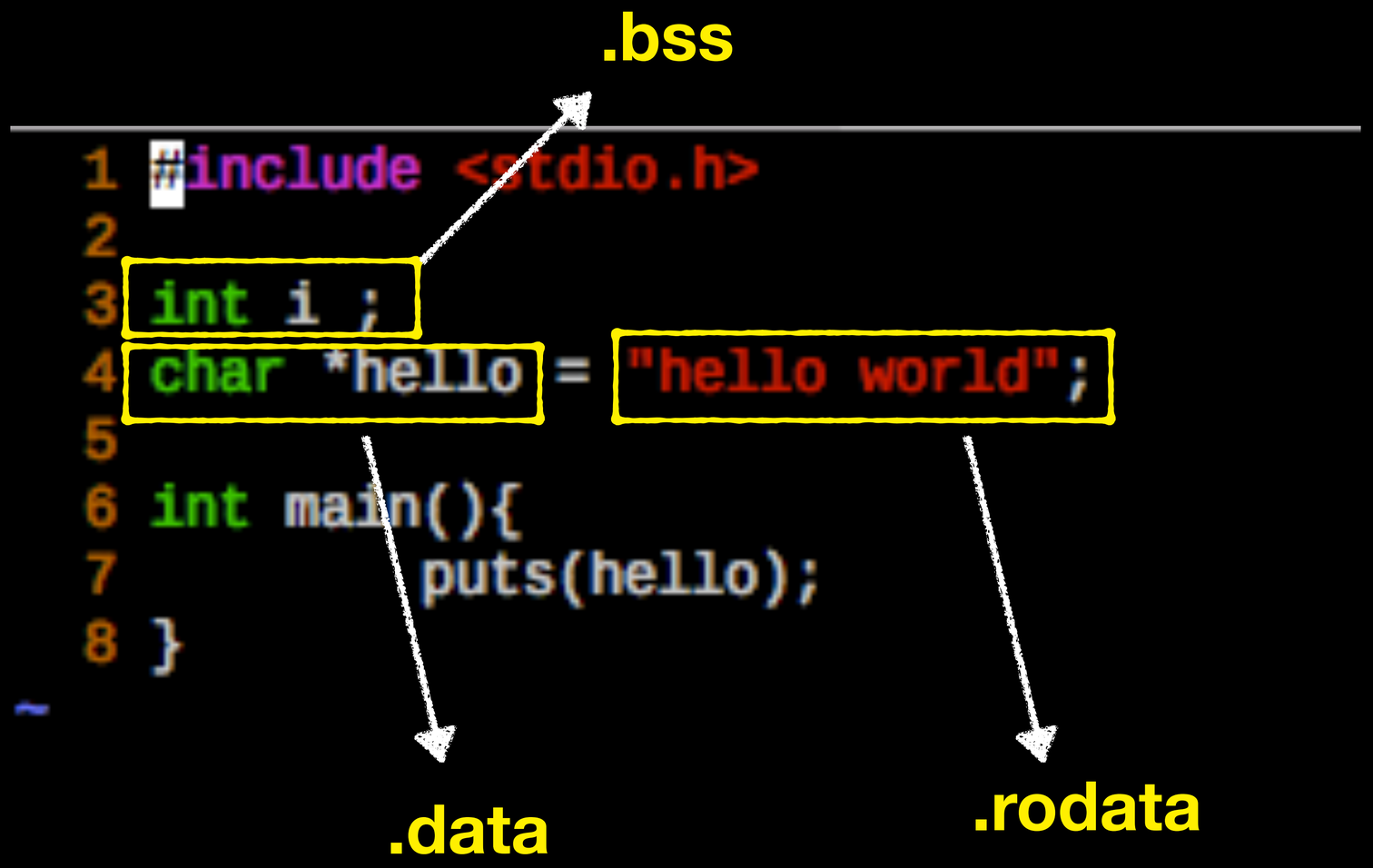
Section

- 在一般情況下程式碼會分成 text、data 以及 bss 等 section ，並不會將 code 跟 data 混在一起使用

Section

- .text
  - 存放 code 的 section
- .data
  - 存放有初始值的全域變數
- .bss
  - 存放沒有初始值的全域變數
- .rodata
  - 存放唯讀資料的 section

Section


.bss
1 #include <stdio.h>
2
3 int i ;
4 char *hello = "hello world";
5
6 int main(){
7 puts(hello);
8 }
~
.data
.rodata

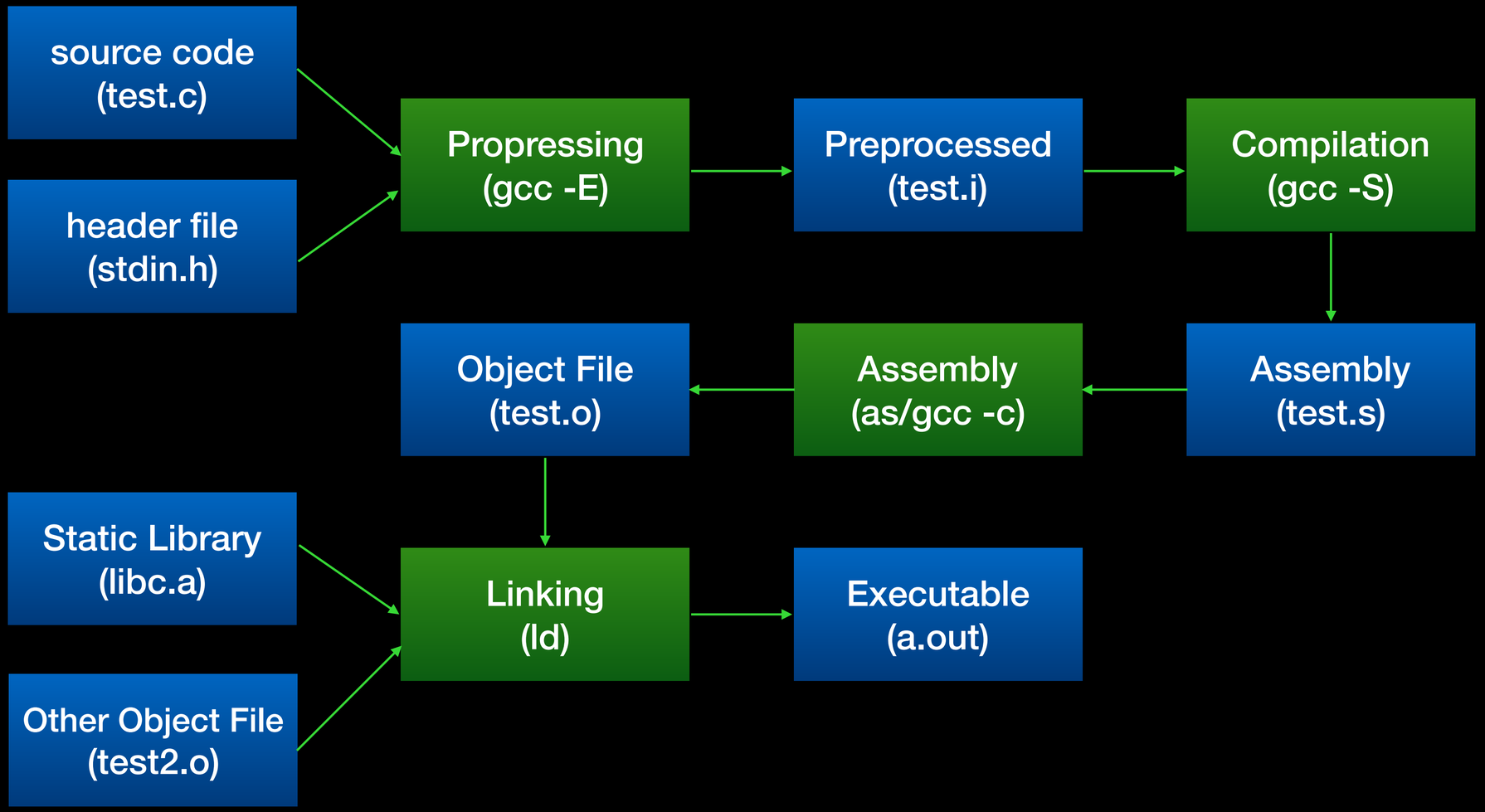
Outline

- Introduction
- Section
- Compilation Flow
- Execution
- x86 assembbly

Compilation flow

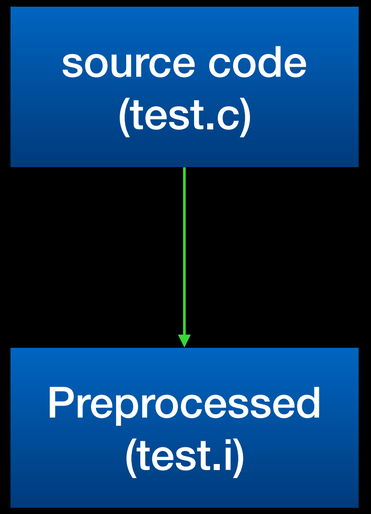
- Preprocessing
- Compilation
- Assembly
- Linking

Compilation flow



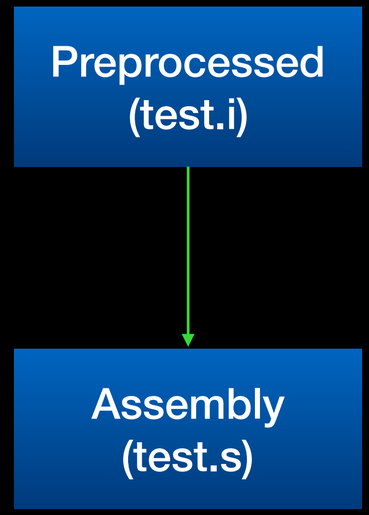
Preprocessing

- 展開 Macro
- 展開 header file
- 刪除所有註解
- 處理所有 preprocess 指令
  - #if 、 #ifnde、#endif …



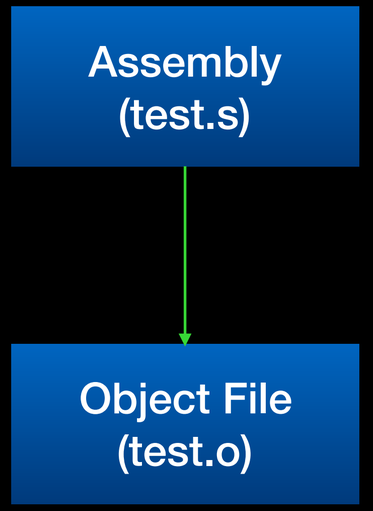
Compilation

- 語法分析 (Syntactic analysis)
- 詞法分析 (Lexical analysis)
- 生成組合語言 (Generate assembly)



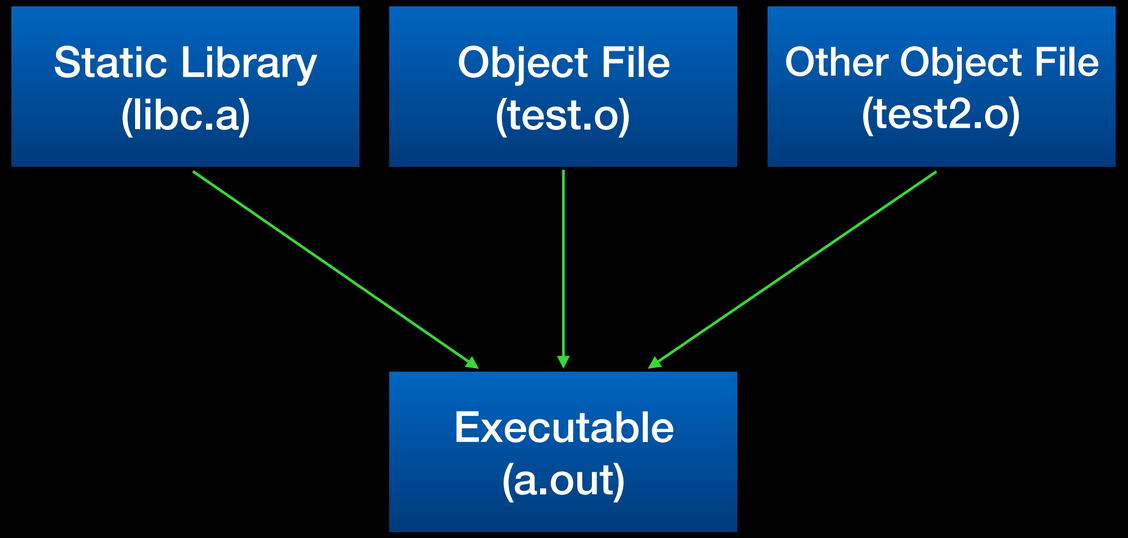
Assembly

- 組合語言到機械碼的過程
- 最後生出 object file
- 此時已是無法用肉眼看懂的 binary

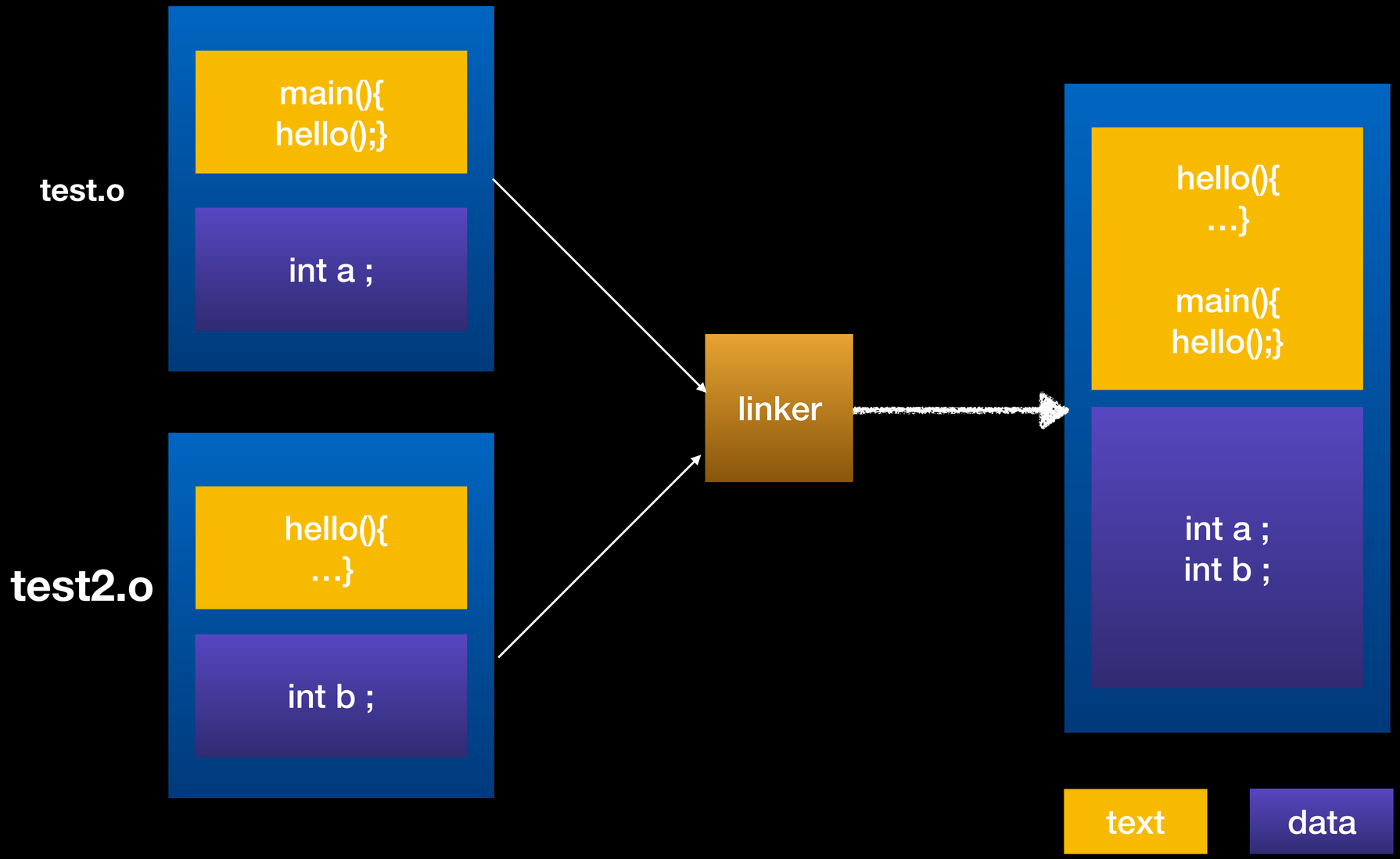


Linking

- Object file 到執行檔的過程 (組裝)
- Relocation

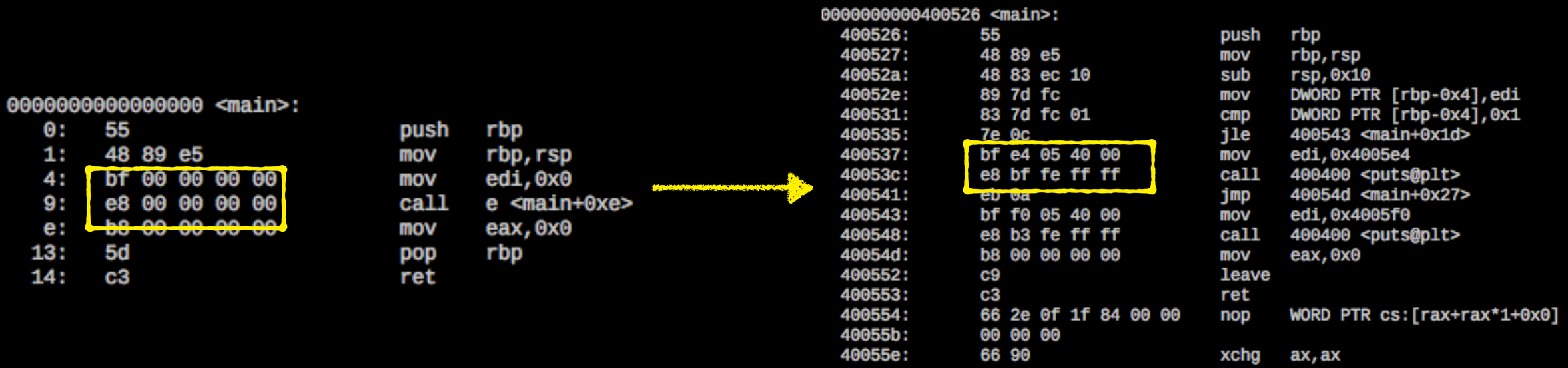


Linking


test.o
main(){
hello();}
int a ;
test2.o
hello(){
…}
int b ;
linker
hello(){
…}
main(){
hello();}
int a ;
int b ;
text
data

Relocation

- 在 object file 生成後，若程式中有 call func 之類的指令，並不會馬上將 address 填入，而是將所有 object file 合併之後，重新定位才將正確位置填入



Static Linking

- 將有用到 library 的 code 也一起編進執行檔中
- 執行檔較肥大
- 一但 library 的 code 有變動需要整個重編

Dynamic Linking

- 所有程式共用一份 library
- 在執行時期才將 library 載入記憶體中
  - 因執行時才去找 function 位置，故執行時間較多
- 執行檔較小
- library 變動不需要重編
- 可能會因為 library 版本不同而行為有所不同

Outline

- Introduction
- Section
- Compilation Flow
- Execution
- x86 assembbly

Execution

- Binary Format
- Segment
- Execution Flow

Binary Format

- 執行檔的格式會根據 OS 不同，而有所不同
  - Linux - ELF
  - Windows - PE
- 在 Binary 的開頭會有個 magic number 欄位，方便讓 OS 辨認是屬於什麼樣類型的檔案
  - 在 Linux 下可以使用 file 來檢視

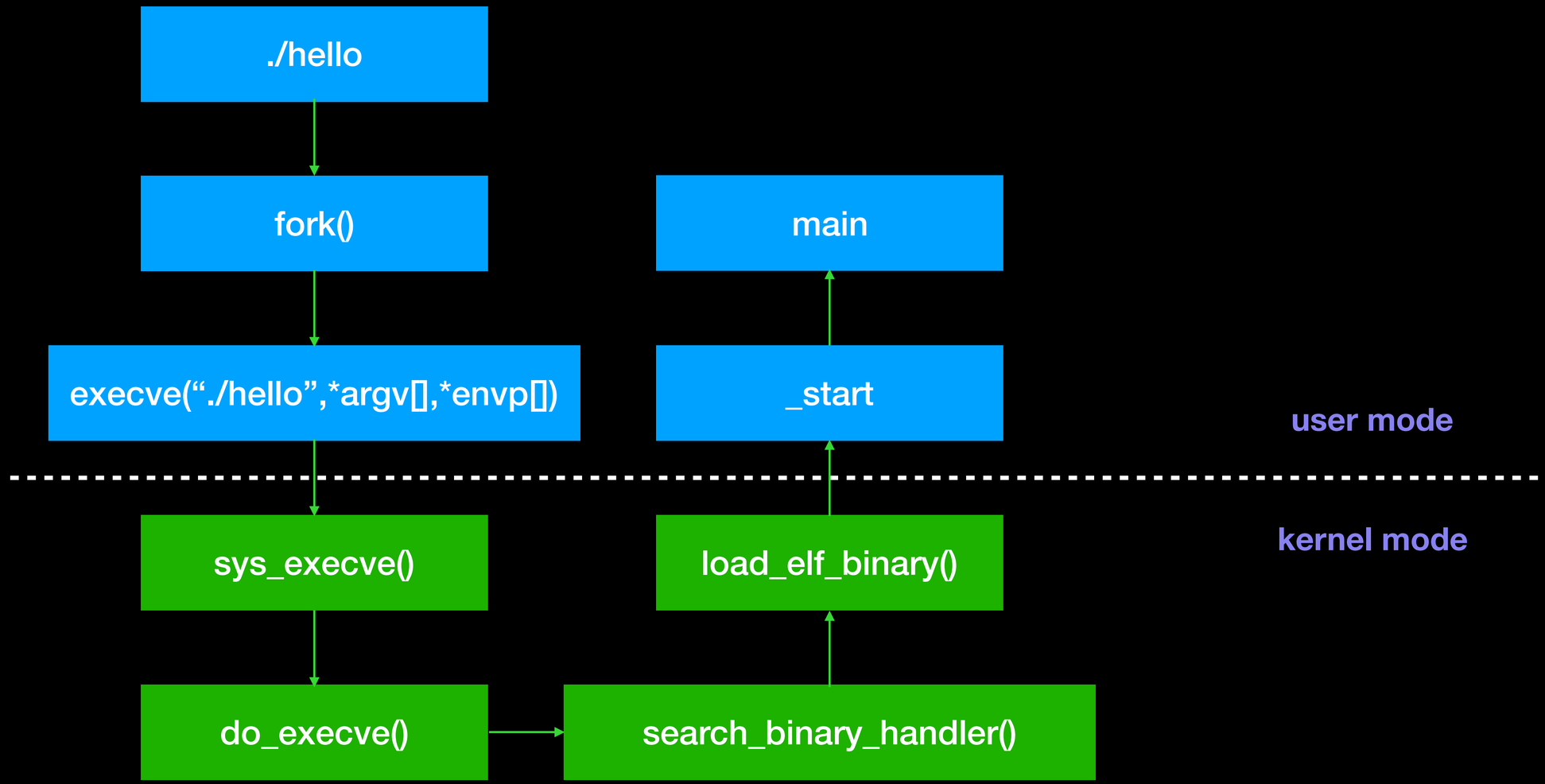
Segment

- 在程式執行時期才會有的概念，基本上會根據讀寫執行權限及特性來分為數個 segment
- 一般來說可分為 rodata、data、code、stack、heap 等 segment
  - data : rw-
  - code : r-x
  - stack : rw-
  - heap : rw-

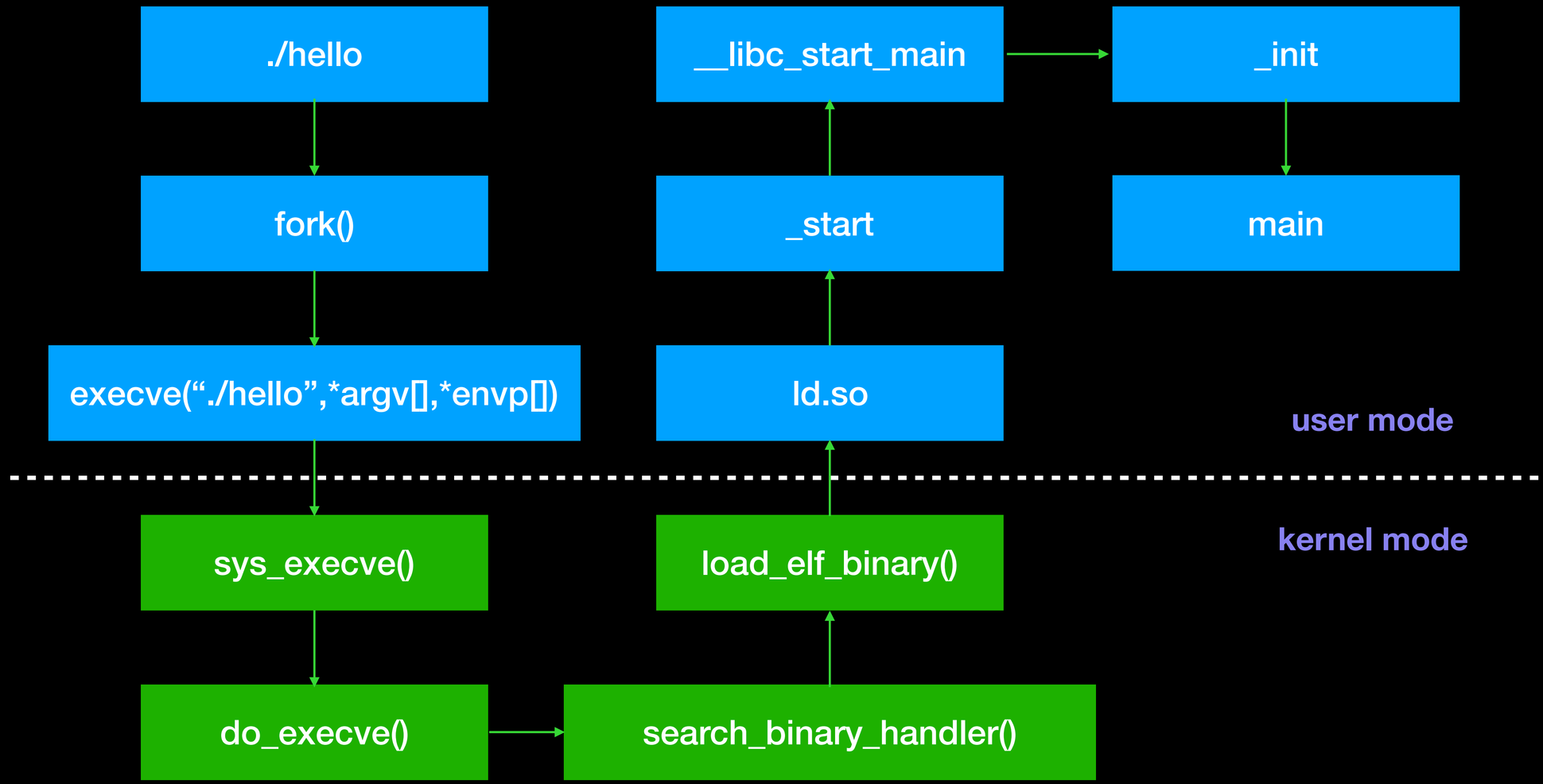
Execution Flow

- What happened when we execute an elf file ?
  - $ ./hello
- 在一般情況下程式會在 disk 中，而 kernel 會通過一連串的過程來將程式 mapping 到記憶體中去執行

Execution Flow

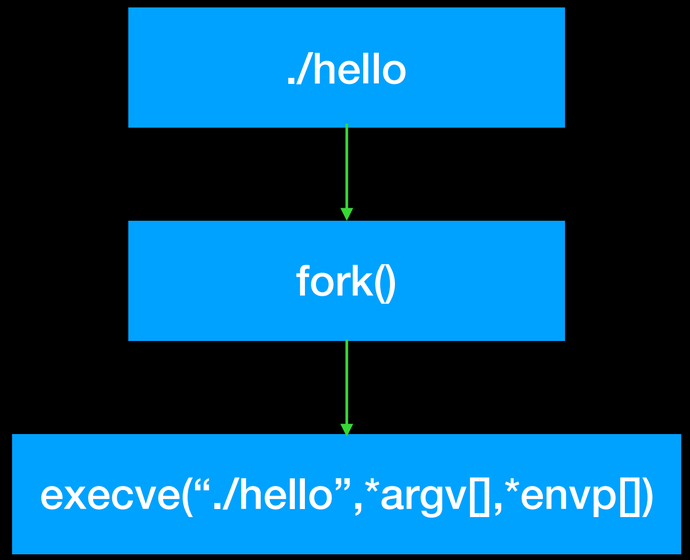


Execution Flow



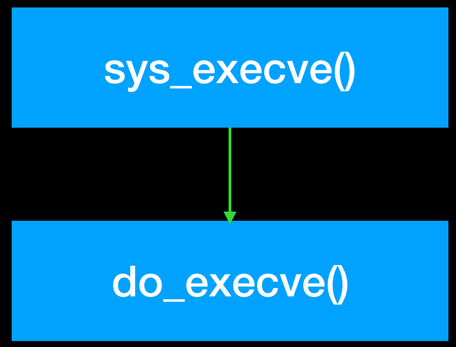
Execution Flow

- 我們在 shell 執行一個 elf 時
  - 會先去 fork 一個 process
  - child 再去使用 execve 執行



Execution Flow

- 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
    - e.g. magic number



Execution Flow

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

search_binary_handler()

Execution Flow

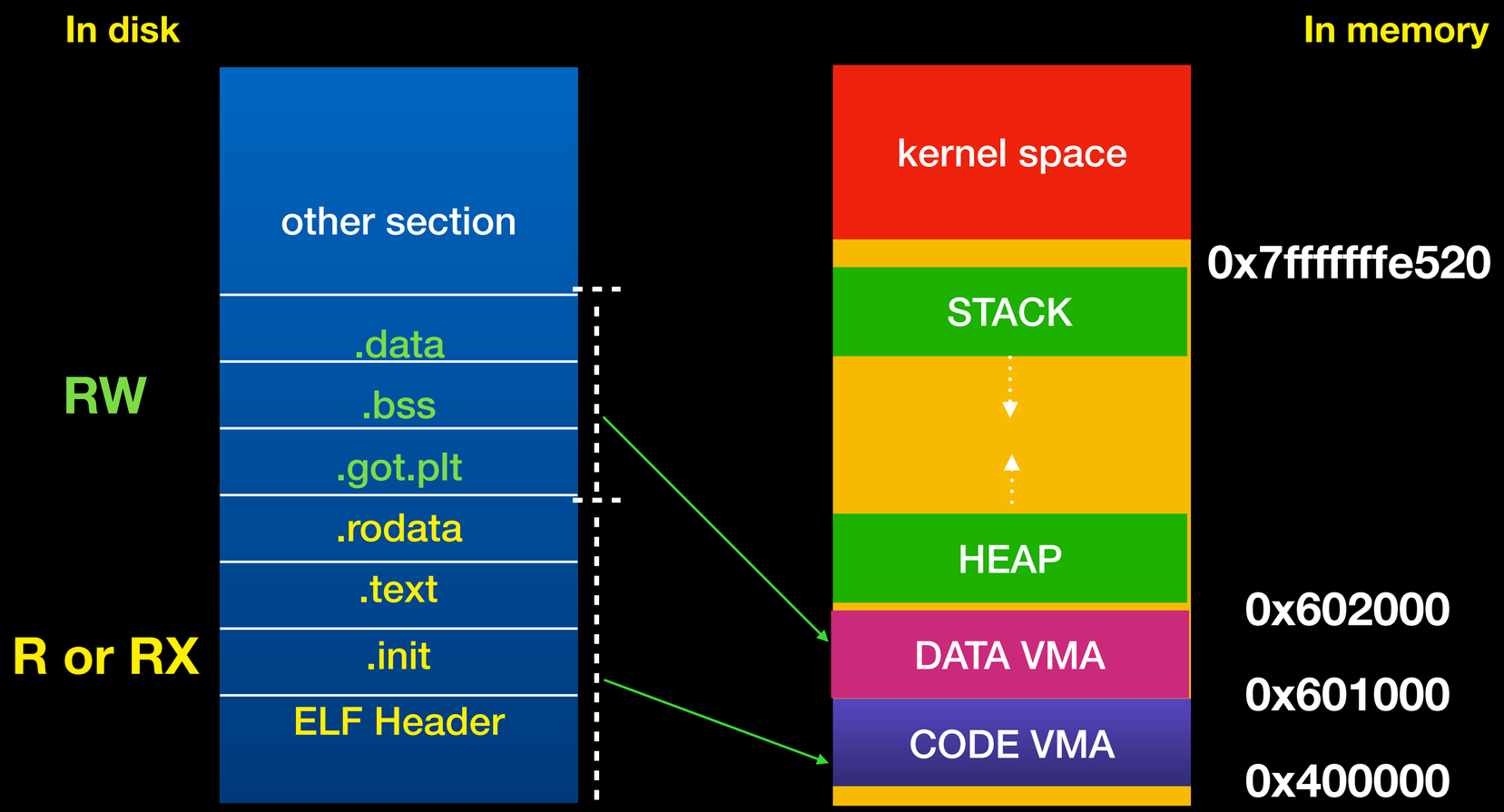
- load_elf_binary()
  - 檢查及獲取 program header 資訊
  - 如果是 dynamic linking 則利用 .interp 這個 section 來確定 loader 路徑
  - 將 program header 紀錄的位置 mapping 到 memory 中，e.g. code segment 位置
  - 將 sys_execve 的 return address 改為 loader (ld.so) 的entry point
    - static linking 下則會是 elf 的 entry point

Execution Flow

- 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
      - 一個 segment 可以包含 0 個到多個 section

Execution Flow

- How program maps to virtual memory.



Execution Flow

- How program maps to virtual memory.
  - readelf -l binary
    - 查看 program header
  - readelf -S binary
    - 查看 section header
  - readelf -d binary
    - 查看 dynamic section 内容

Execution Flow

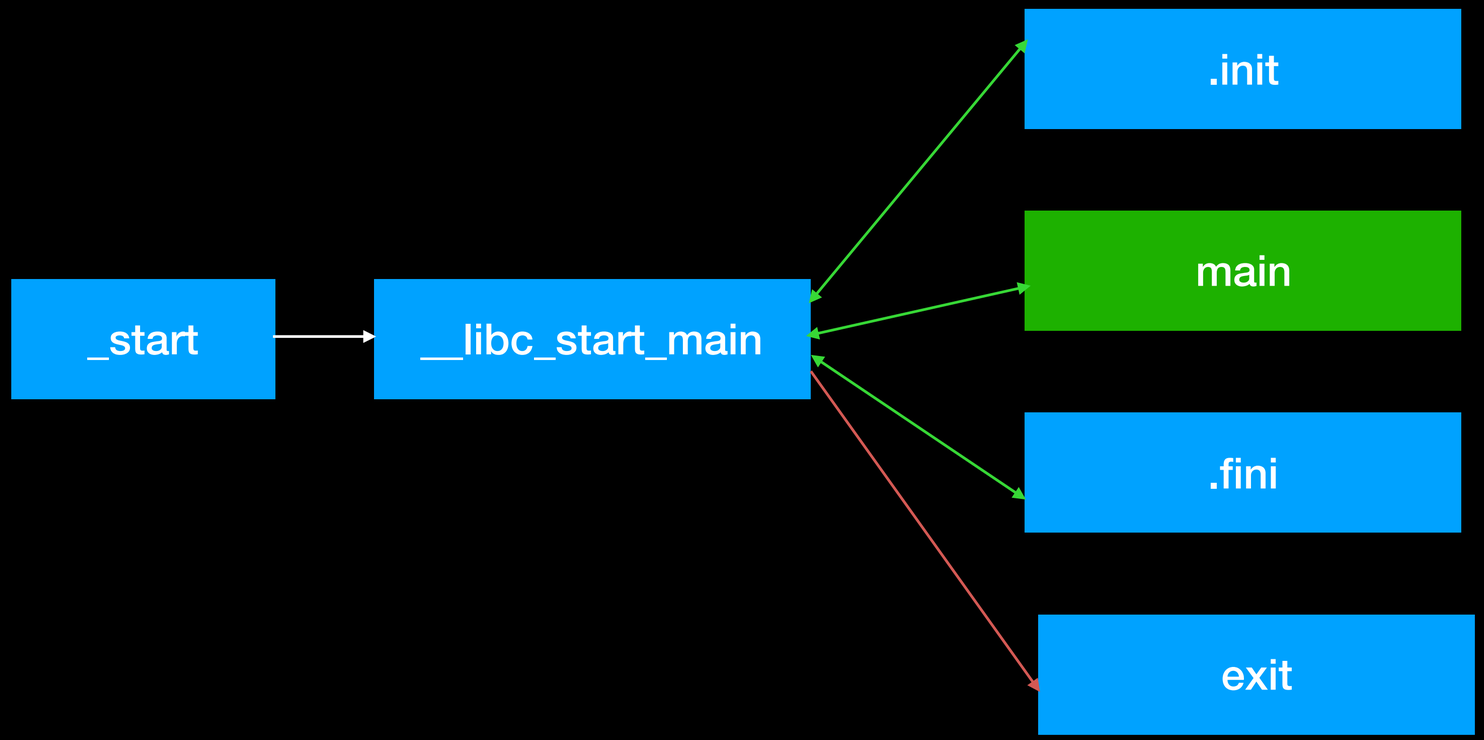
- How program maps to virtual memory.

Execution flow

- 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
  - 對實際運作過程有興趣可參考 elf/rtld.c

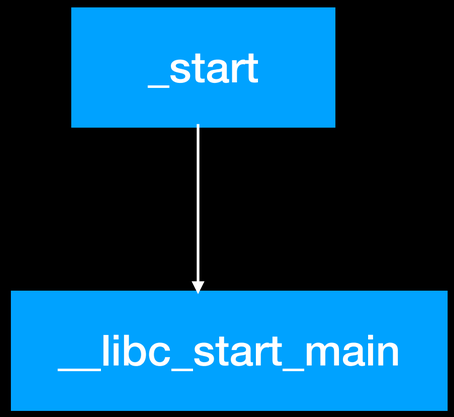
Execution Flow

- 在 ld.so 執行完後會跳到 _start 開始執行主要程式



Execution Flow

- _start
  - 將下列項目傳給 libc_start_main
    - 環境變數起始位置
    - main 的位置 (通常在第一個參數)
    - .init
      - 呼叫 main 之前的初始化工作
    - .fini
      - 程式結束前的收尾工作



Execution Flow

- _libc_start_main
  - 執行 .init
  - 執行 main
    - 主程式部分
  - 執行 .fini
  - 執行 exit 結束程式

Outline

- Introduction
- Section
- Compilation Flow
- Execution
- x64 assembly

x64 assembly

- Registers
  - General-purpose registers
    - RAX RBX RCX RDX RSI RDI- 64 bit
    - EAX EBX ECX EDX ESI EDI - 32 bit
    - AX BX CX DX SI DI - 16 bit

x64 assembly

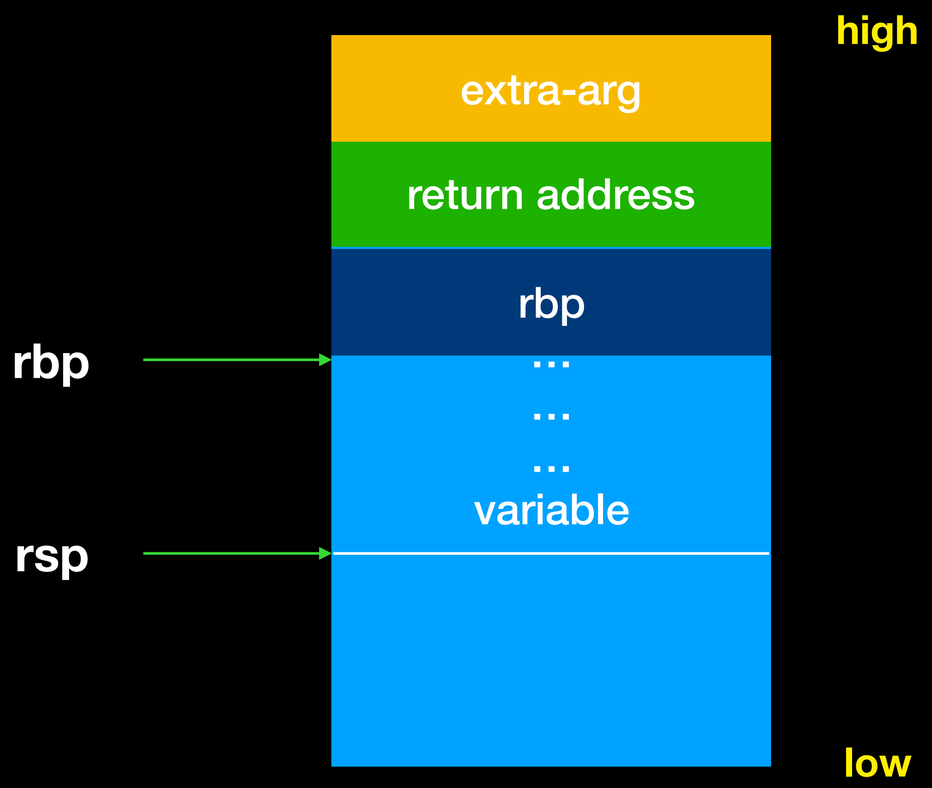
- Registers
  - r8 r9 r10 r11 r12 r13 r14 r15 - 64 bit
  - r8d r9d r10d … - 32 bit
  - r8w r9w r10w … -16 bit
  - r8b r9b r10b … - 8 bit

x64 assembly

- Registers
  - Stack Pointer Register
    - RSP
  - Base Pointer Register
    - RBP
  - Program Counter Register
    - RIP

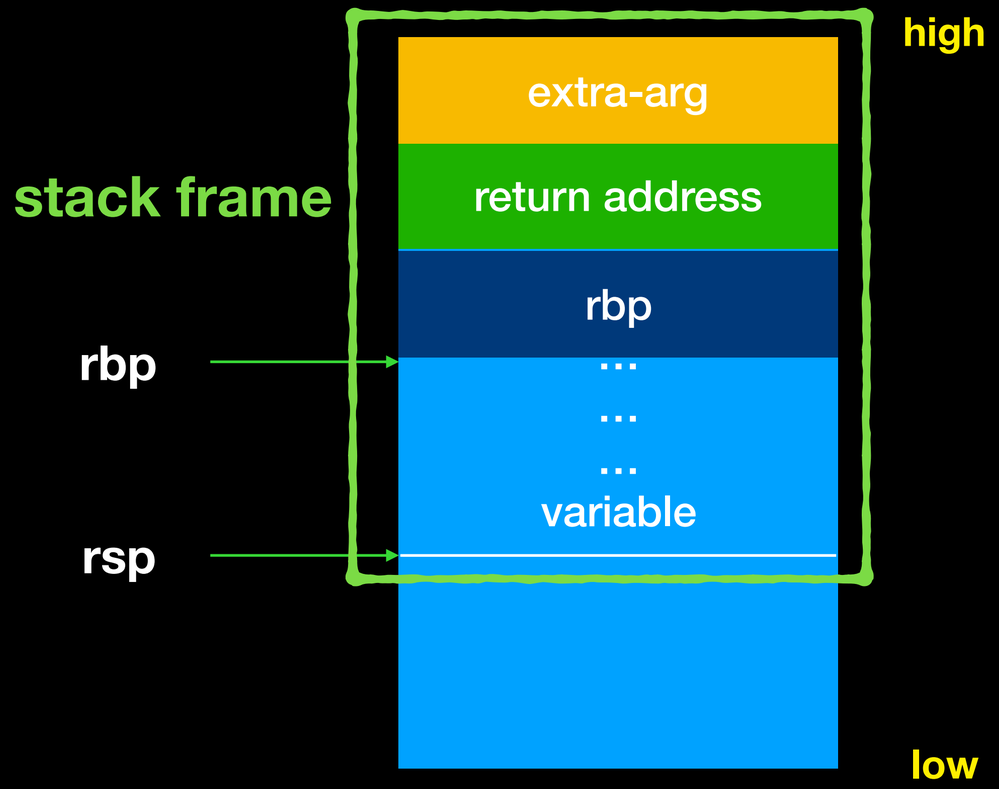
x64 assembly

- Registers
  - Stack Pointer
    - RSP - 64 bit
    - 指向 stack 頂端
  - Base Pointer
    - RBP - 64 bit
    - 指向 stack 底端
  - RSP 到 function 參數範圍稱為該 function 的 Stack Frame



x64 assembly

- Registers
  - Stack Pointer
    - RSP - 64 bit
    - 指向 stack 頂端
  - Base Pointer
    - RBP - 64 bit
    - 指向 stack 底端
  - RSP 到 function 參數範圍稱為該 function 的 Stack Frame



x64 assembly

- Registers
  - Program counter register
    - RIP
    - 指向目前程式執行的位置
  - Flag register
    - eflags
    - 儲存指令執行結果
  - Segment register
    - cs ss ds es fs gs

x64 assembly

- AT & T
  - mov %rax,%rbx
- Intel
  - mov rbx,rax

x64 assembly

- Basic instruction
  - mov
  - add/sub
  - and/or/xor
  - push/pop
  - lea
  - jmp/call/ret

x64 assembly

- mov
  - mov imm/reg/mem value to reg/mem
  - mov A,B (move B to A)
  - A 與 B 的 size 要相等
  - ex :
    - mov rax,rbx (o)
    - mov rax,bx (x)
    - mov rax,0xdeadbeef

x64 assembly

- add/sub/or/xor/and
  - add/sub/or/xor/and reg,imm/reg
  - add/sub/or/xor/and A,B
  - A 與 B 的 size 一樣要相等
  - ex :
    - add rbp,0x48
    - sub rax,rbx

x64 assembly

- push/pop
  - push/pop reg
  - ex :
    - push rax = sub rsp,8 ; mov [rsp],eax
    - pop rbx = mov rbx,[rsp] ; add rsp,8

x64 assembly

- lea
  - ex :
    - lea rax, [rsp+8]

x64 assembly

- lea v.s. mov
  - lea rax, [rsp+8] v.s mov rax,[rsp+8]
  - assume
    - rax = 3
    - rsp+8 = 0x7ffffffe4c0
    - [rsp+8] = 0xdeadbeef

lea

rax = 0x7ffffffe4c0

mov

rax = 0xdeadbeef

x64 assembly

- jmp/call/ret
  - jmp 跳至程式碼的某處去執行
  - call rax = push rip+8 ;jmp rax
  - ret = pop rip

x64 assembly

- leave
  - mov rsp,rbp
  - pop rbp

x64 assembly

- nop
  - 一個 byte 不做任何事
  - opcode = 0x90

x64 assembly

- System call
  - Instruction : syscall
  - SYSCALL NUMBER: RAX
  - Argument : RDI RSI RDX RCX R8 R9
  - Return value : RAX

x64 assembly

- system call table
- https://w3challs.com/syscalls/?arch=x86_64

Show 10 entries

#	Name	Registers			
		rax	rdi	rsi	rdx
0	read	0x00	unsigned int fd	char *buf	size_t count
1	write	0x01	unsigned int fd	const char *buf	size_t count
2	open	0x02	const char *filename	int flags	umode_t mode
3	close	0x03	unsigned int fd	–	–
4	stat	0x04	const char *filename	struct __old_kernel_stat *statbuf	–
5	fstat	0x05	unsigned int fd	struct __old_kernel_stat *statbuf	–
6	lstat	0x06	const char *filename	struct __old_kernel_stat *statbuf	–
7	poll	0x07	struct pollfd *ufds	unsigned int nfds	int timeout_msecs
8	lseek	0x08	unsigned int fd	off_t offset	unsigned int origin
9	mmap	0x09	unsigned long addr	unsigned long len	unsigned long prot
10	mprotect	0x0a	unsigned long start	size_t len	unsigned long prot
11	munmap	0x0b	unsigned long addr	size_t len	–
12	brk	0x0c	unsigned long brk	–	–
13	rt_sigaction	0x0d	int sig	const struct sigaction *act	struct sigaction *oact
14	rt_sigprocmask	0x0e	int how	sigset_t *nset	sigset_t *oset
15	rt_sigreturn	0x0f	–	–	–
16	ioctl	0x10	unsigned int fd	unsigned int cmd	unsigned long arg
17	pread64	0x11	char *buf size_t count	loff_t pos	–

x64 assembly

- Calling convention
  - function call
    - call : push return address to stack then jump
  - function return
    - ret ：pop return address
  - function argument
    - 基本上用 register 傳遞
      - 依序為 rdi rsi rdx r10 r8 r9
    - 依序放到 register，再去執行 function call

x64 assembly

- Calling convention
  - function prologue
    - compiler 在 function 開頭加的指令，主要在保存 rbp 分配區域變數所需空間

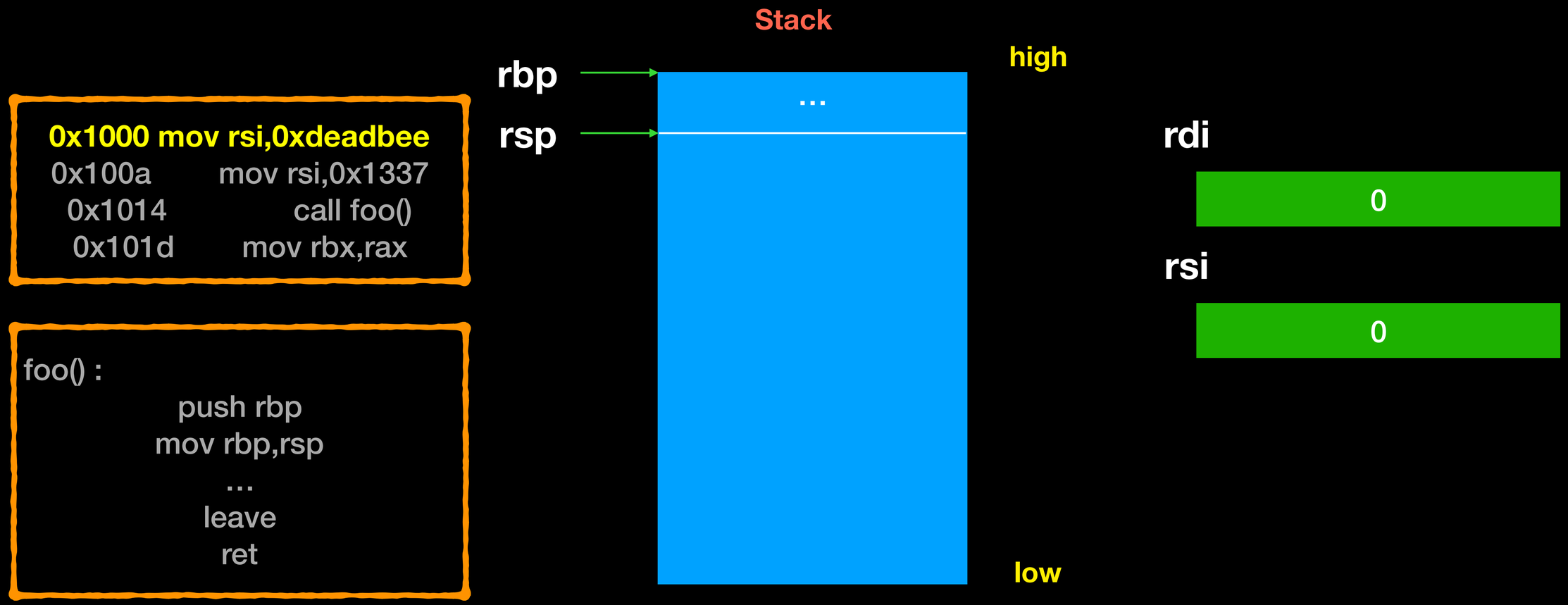
```
push rbp
mov rbp,rsp
sub rsp,0x30
```

x64 assembly

- Calling convention
  - function epilogue
    - compiler 在 function 結尾加的指令，主要在利用保存的 rbp 回覆 call function 時的 stack 狀態

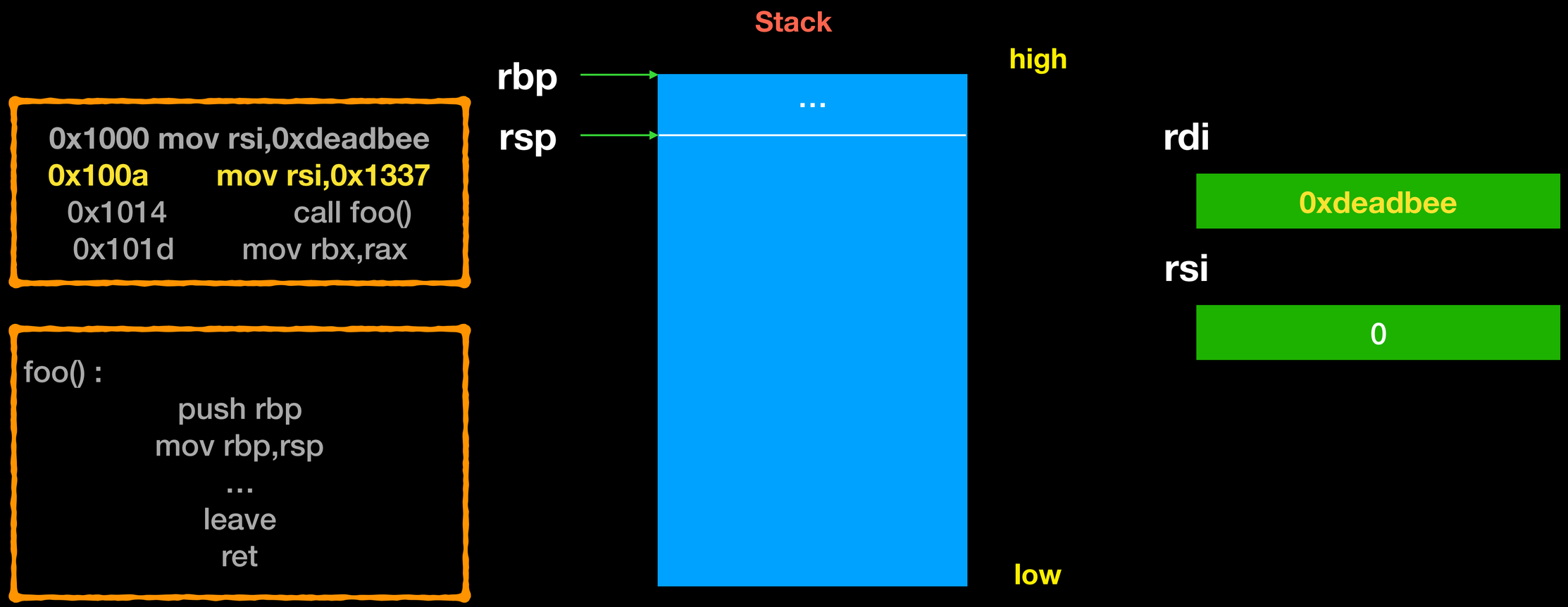
x64 assembly

- Calling convention
  - call foo(0x1337,0xdeadbee)



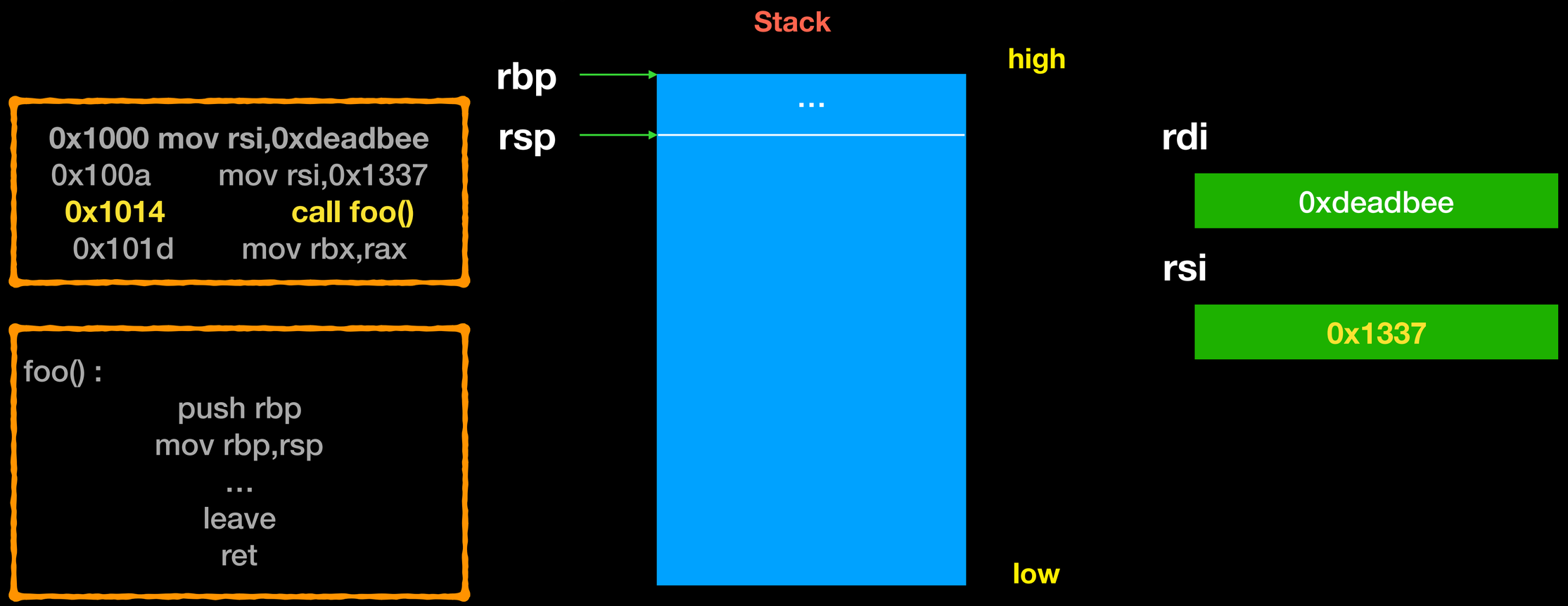
x64 assembly

- Calling convention
  - call foo(0x1337,0xdeadbee)



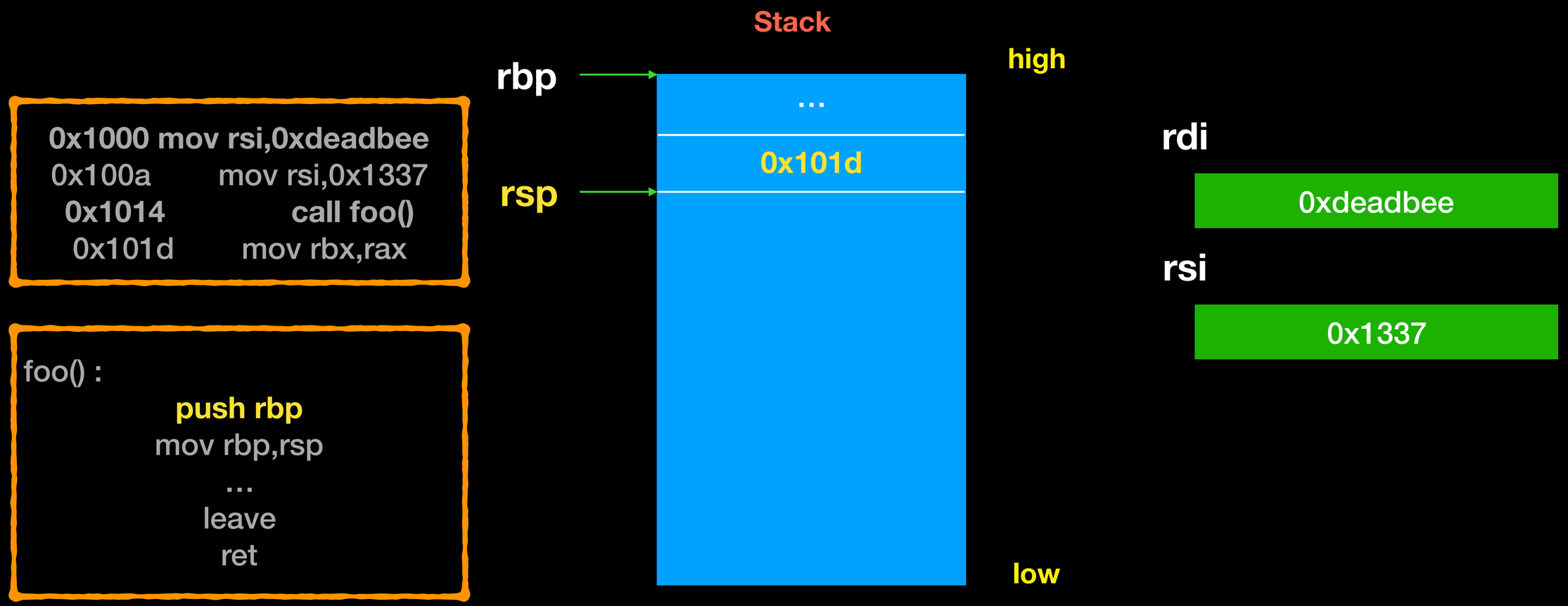
x64 assembly

- Calling convention
  - call foo(0x1337,0xdeadbee)



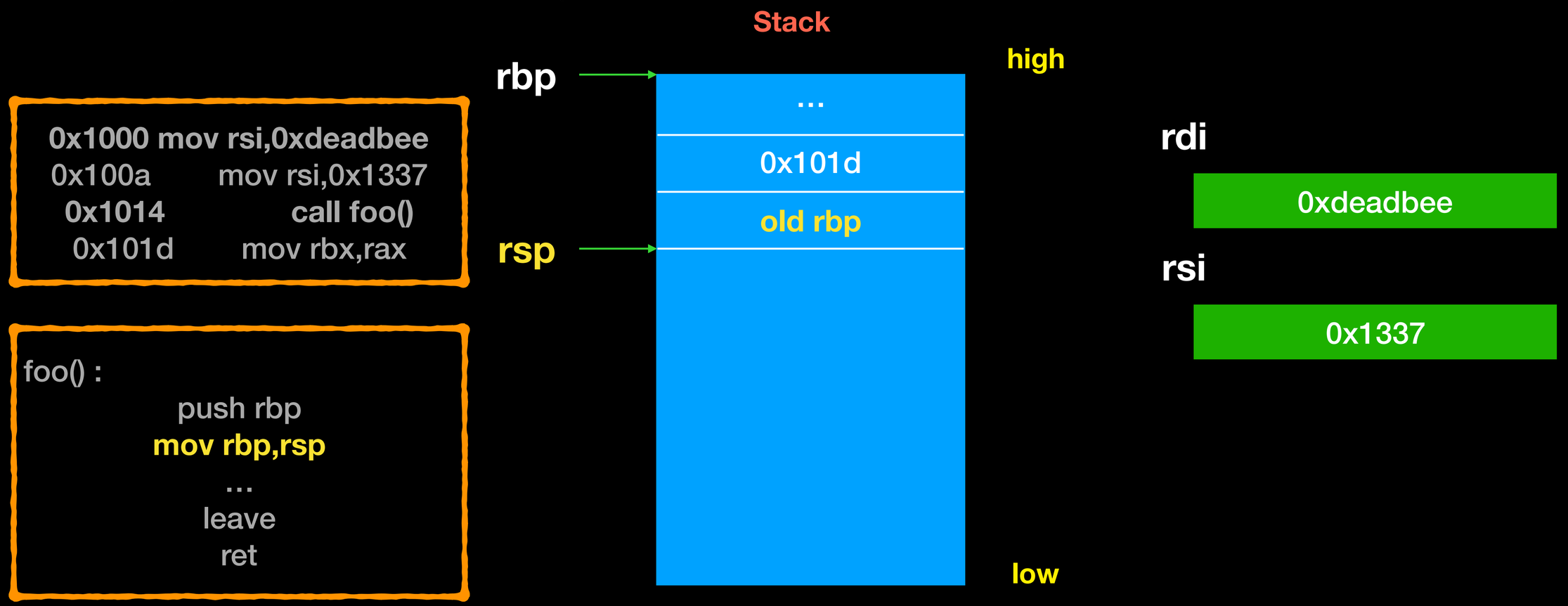
x64 assembly

- Calling convention
  - call foo(0x1337,0xdeadbee)



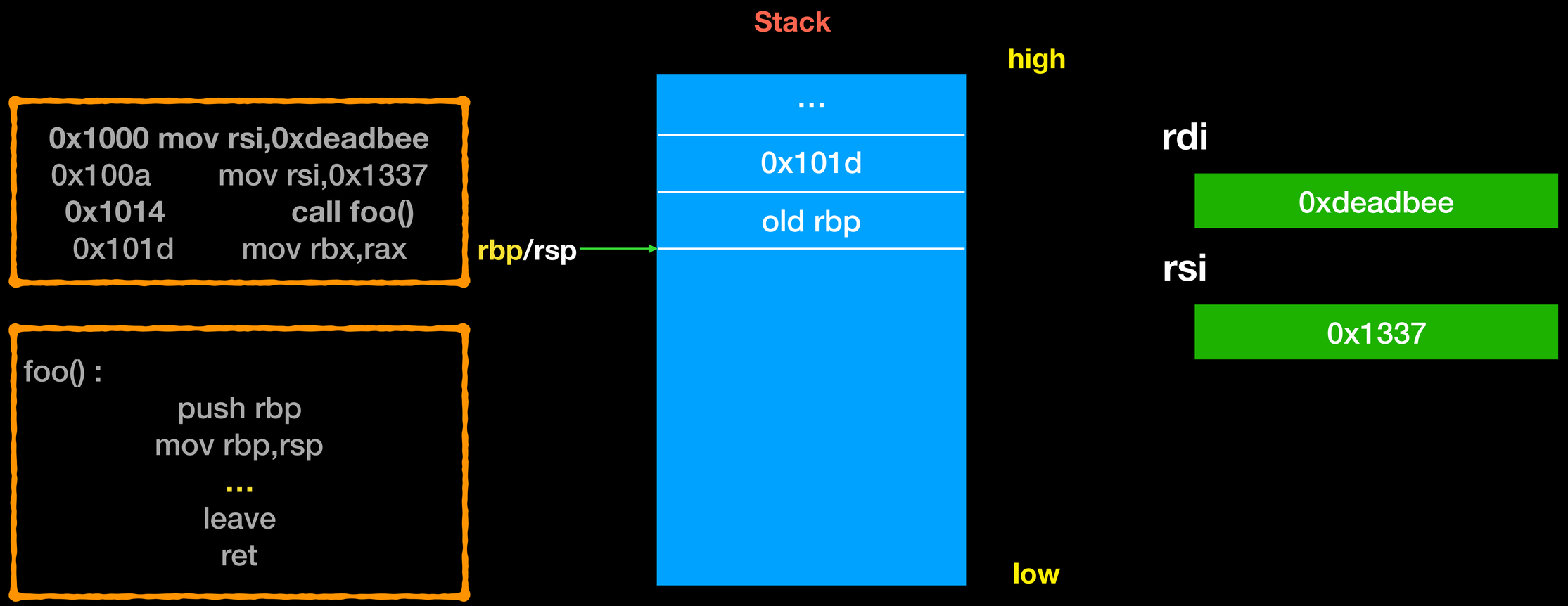
x64 assembly

- Calling convention
  - call foo(0x1337,0xdeadbee)



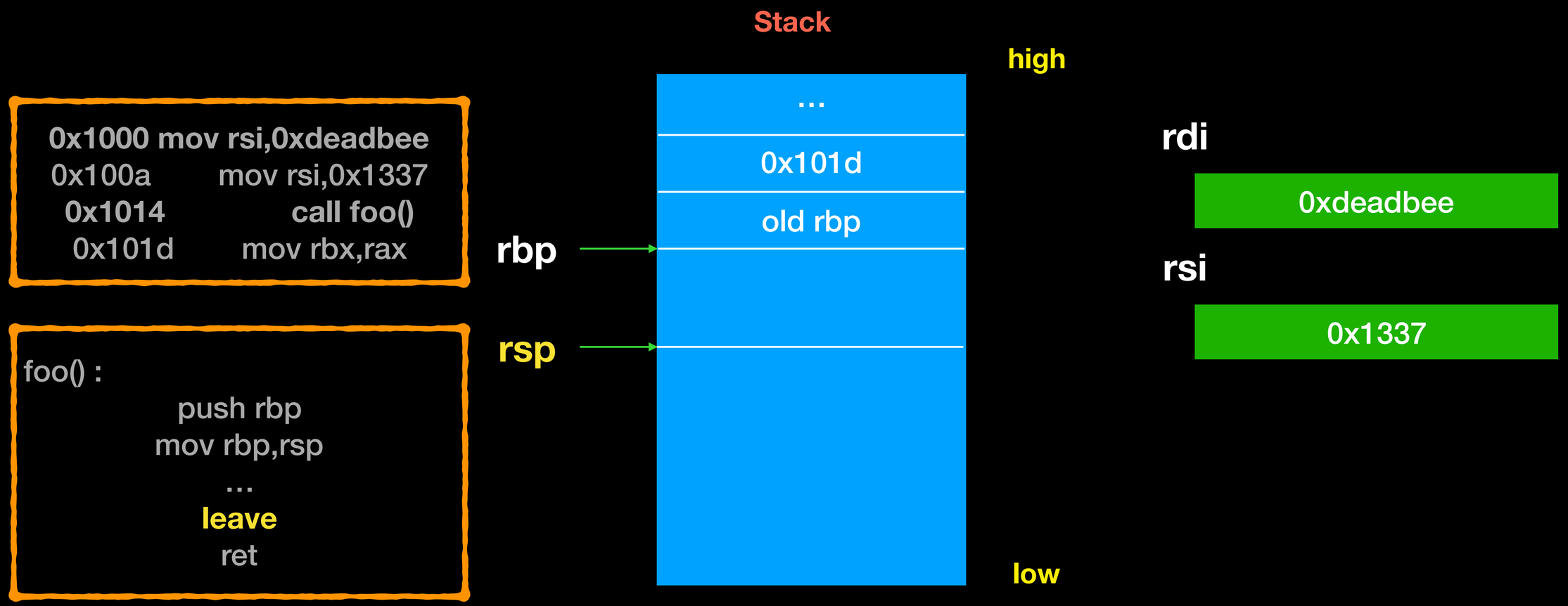
x64 assembly

- Calling convention
  - call foo(0x1337,0xdeadbee)



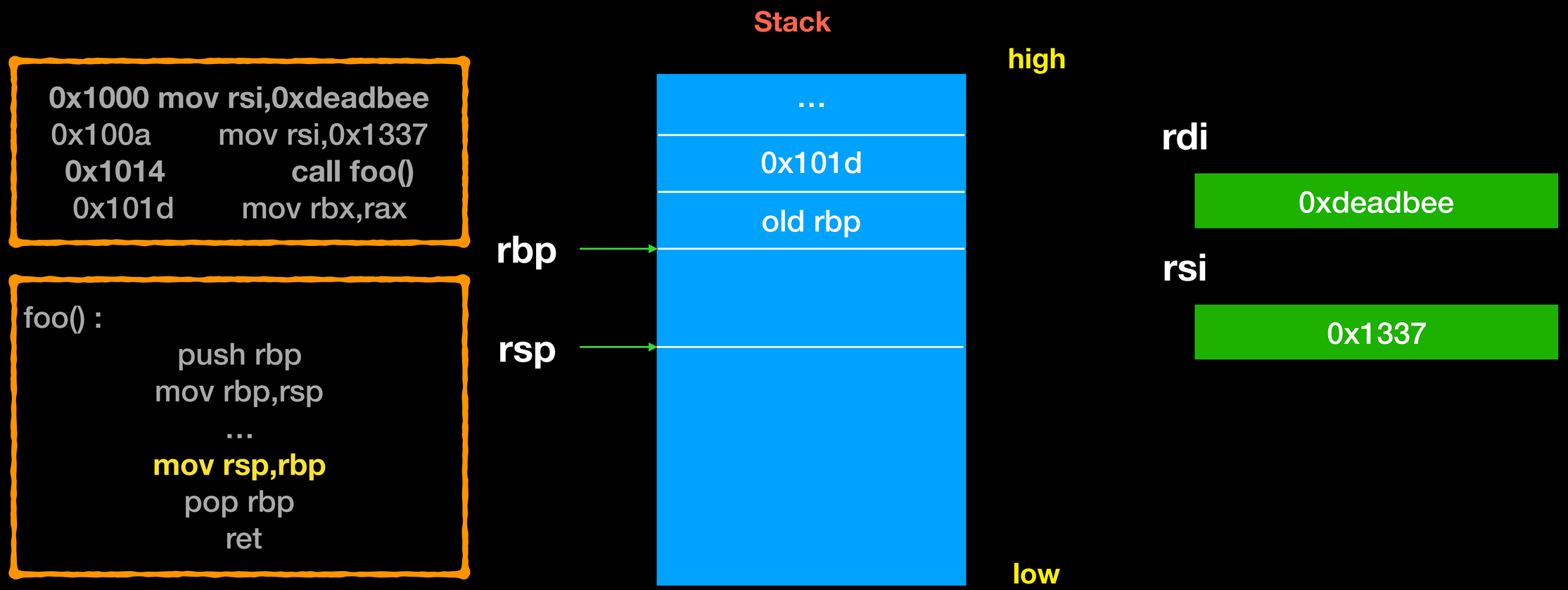
x64 assembly

- Calling convention
  - call foo(0x1337,0xdeadbee)



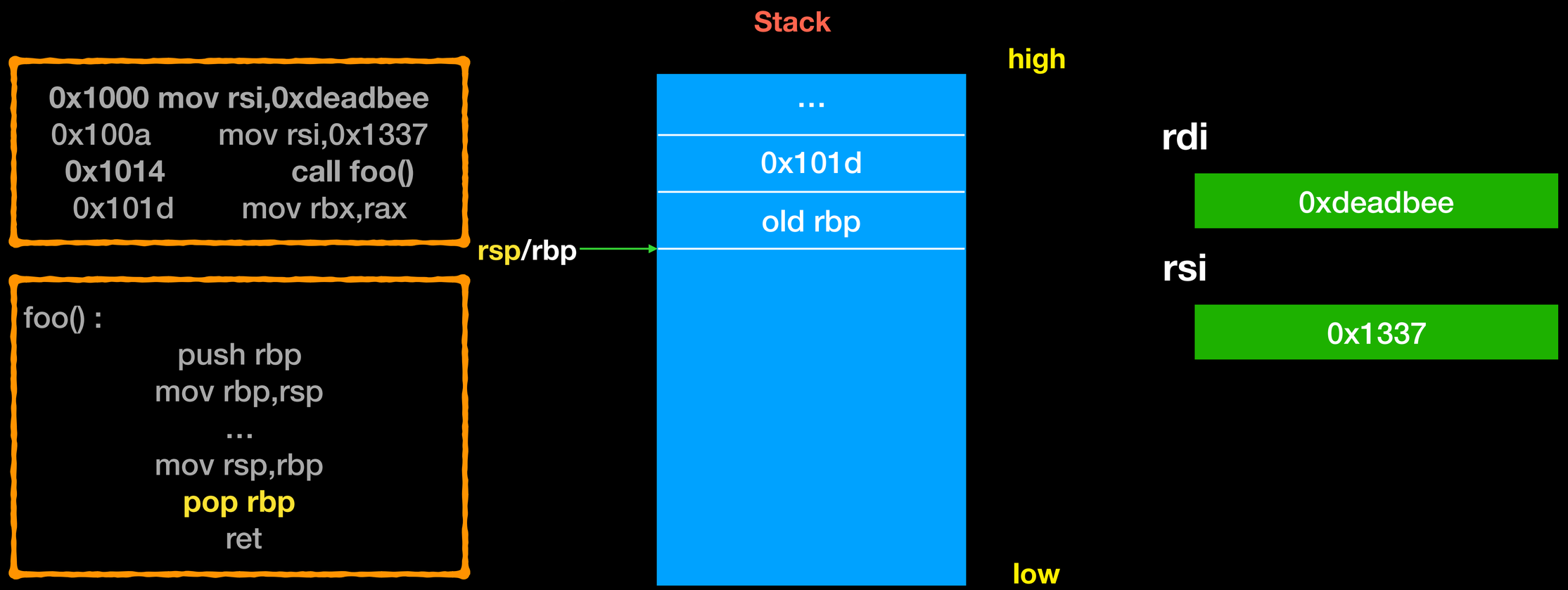
x64 assembly

- Calling convention
  - call foo(0x1337,0xdeadbee)



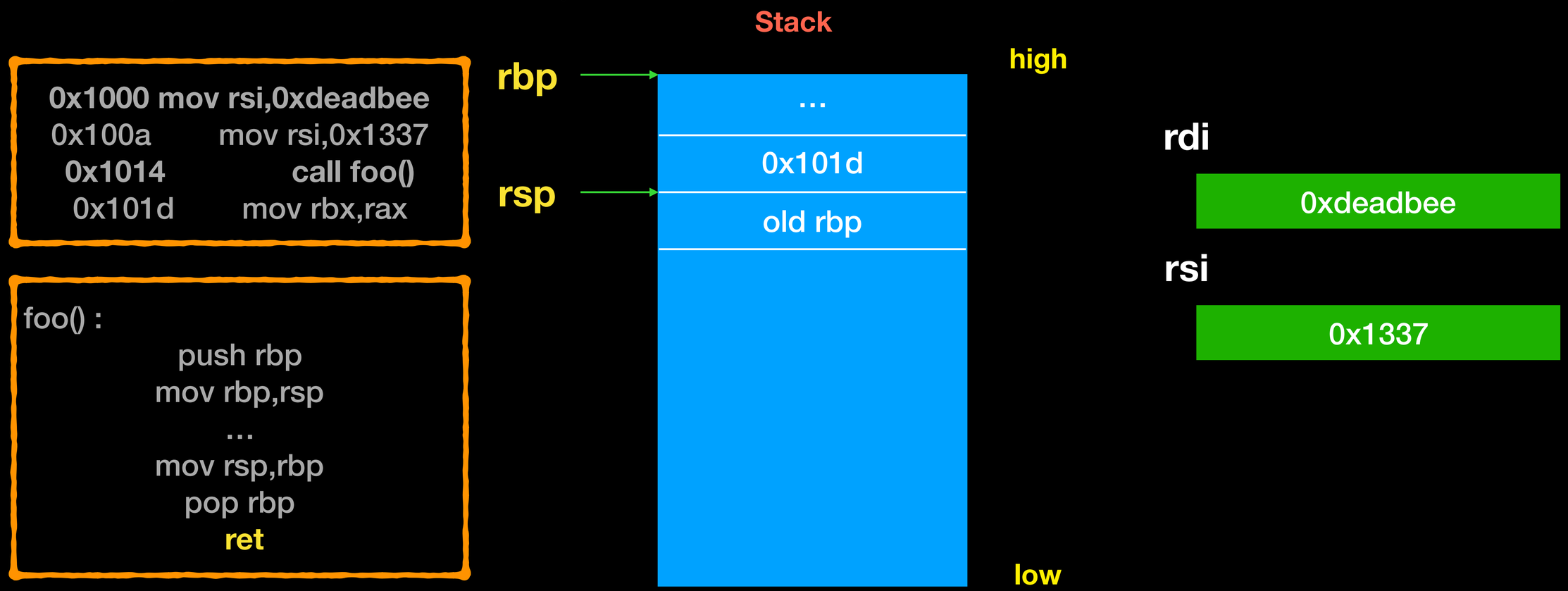
x64 assembly

- Calling convention
  - call foo(0x1337,0xdeadbee)



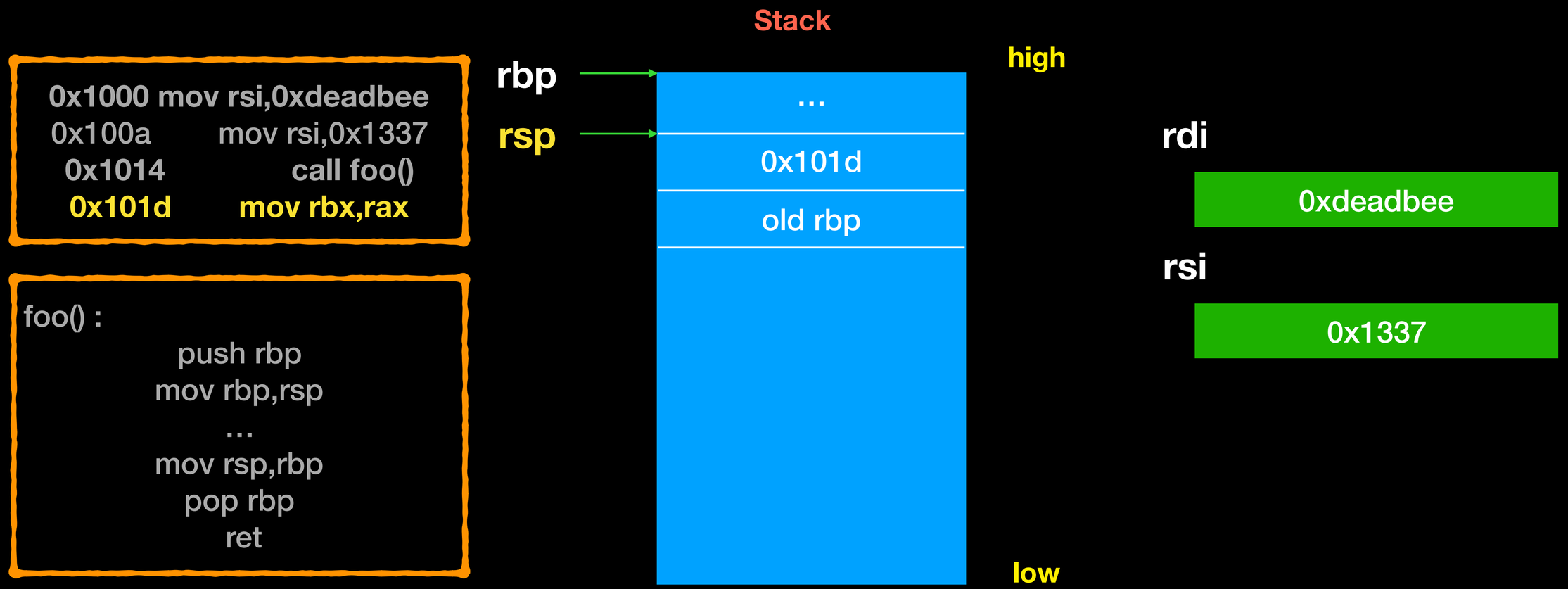
x64 assembly

- Calling convention
  - call foo(0x1337,0xdeadbee)



x64 assembly

- Calling convention
  - call foo(0x1337,0xdeadbee)



x64 assembly

- Hello world
  - nasm -felf64 hello.s -o hello.o
  - ld -m elf_x86_64 hello.o -o hello

```
1  global _start
2
3  section .text
4  _start :
5      xor rax,rax
6      xor rbx,rbx
7      xor rcx,rcx
8      xor rdx,rdx
9      jmp str
10 write :
11     mov rax,1 ;write
12     inc rdi
13     pop rsi
14     mov rdx,12
15     syscall
16
17     mov rax,60 ;exit
18     syscall
19
20 str :
21     call write
22     db 'Hello world',0
23
```

x64 assembly

- Shellcode
  - 顧名思義，攻擊者主要注入程式碼後的目的為拿到 shell ，故稱 shellcode
  - 由一系列的 machine code 組成，最後目的可做任何攻擊者想做的事

x64 assembly

- Hello world shellcode

```
0000000000400080 <_start>:
  400080:       48 31 c0                xor    rax,rax
  400083:       48 31 db                xor    rbx,rbx
  400086:       48 31 c9                xor    rcx,rcx
  400089:       48 31 d2                xor    rdx,rdx
  40008c:       eb 17                   jmp    4000a5 <str>

000000000040008e <write>:
  40008e:       b8 01 00 00 00          mov    eax,0x1
  400093:       48 ff c7                inc    rdi
  400096:       5e                      pop    rsi
  400097:       ba 0c 00 00 00          mov    edx,0xc
  40009c:       0f 05                   syscall
  40009e:       b8 3c 00 00 00          mov    eax,0x3c
  4000a3:       0f 05                   syscall

00000000004000a5 <str>:
  4000a5:       e8 e4 ff ff ff          call   40008e <write>
  4000aa:       68 65 6c 6c 6f          push   0x6f6c6c65
  4000af:       20 77 6f                and    BYTE PTR [rdi+0x6f],dh
  4000b2:       72 6c                   jb     400120 <str+0x7b>
  4000b4:       64                      fs
```

shellcode[] = “\x48\x31\xc0\x48\x31\xdb\x48\x31\xc9\x48\x31\xd2\xeb\x17…….\x7x\x6c\x64”

x64 assembly

- 產生 shellcode
  - objcopy -O binary hello.bin shellcode.bin

x64 assembly

- Using Pwntool
  - http://docs.pwntools.com/en/stable/asm.html
- Pwntool bunutils
  - http://docs.pwntools.com/en/stable/install/binutils.html

x64 assembly

- pwn.asm

```
1 #!/usr/bin/env python
2 # -*- coding: utf-8 -*-
3 from pwn import *
4
5 context.arch = "amd64"
6 s = asm("""
7     xor rax,rax
8     xor rdi,rdi
9     xor rsi,rsi
10    xor rdx,rdx
11    jmp getstr
12 write :
13    pop rsi
14    mov rax,1
15    mov rdi,1
16    mov rdx,12
17    syscall
18
19    mov rax,0x3c
20    syscall
21
22 getstr :
23    call write
24    .ascii "hello world"
25    .byte 0
26 """)
```

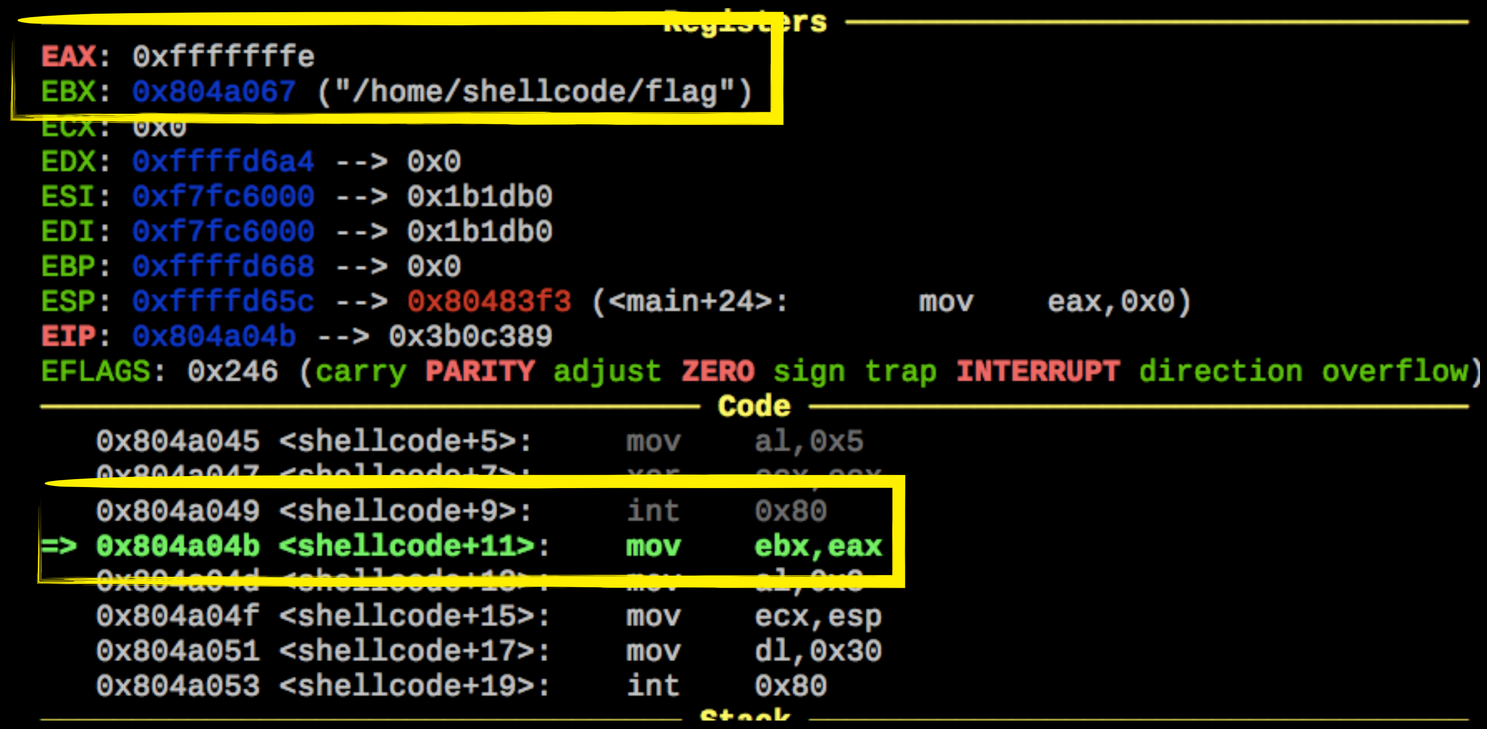
x64 assembly

- Test your shellcode
  - *gcc -z execstack test.c -o test*

```c
#include <stdio.h>

char shellcode[] = "\xeb\x19\x59\xb8\x04\x00\x00\x00\xbb\x01\x00\x00\x00\xba\x0c\x00\x00\x00\xcd\x80\xb8\x01\x00\x00\x00\xcd\x8

int main(){
    void (*fptr)() = shellcode;
    fptr();

}
```

x64 assembly

- How to debug your shellcode
  - *gdb ./test*



Reference

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

Q & A