Reverse Engineering - 1

2021/11/19

Presented by LJP

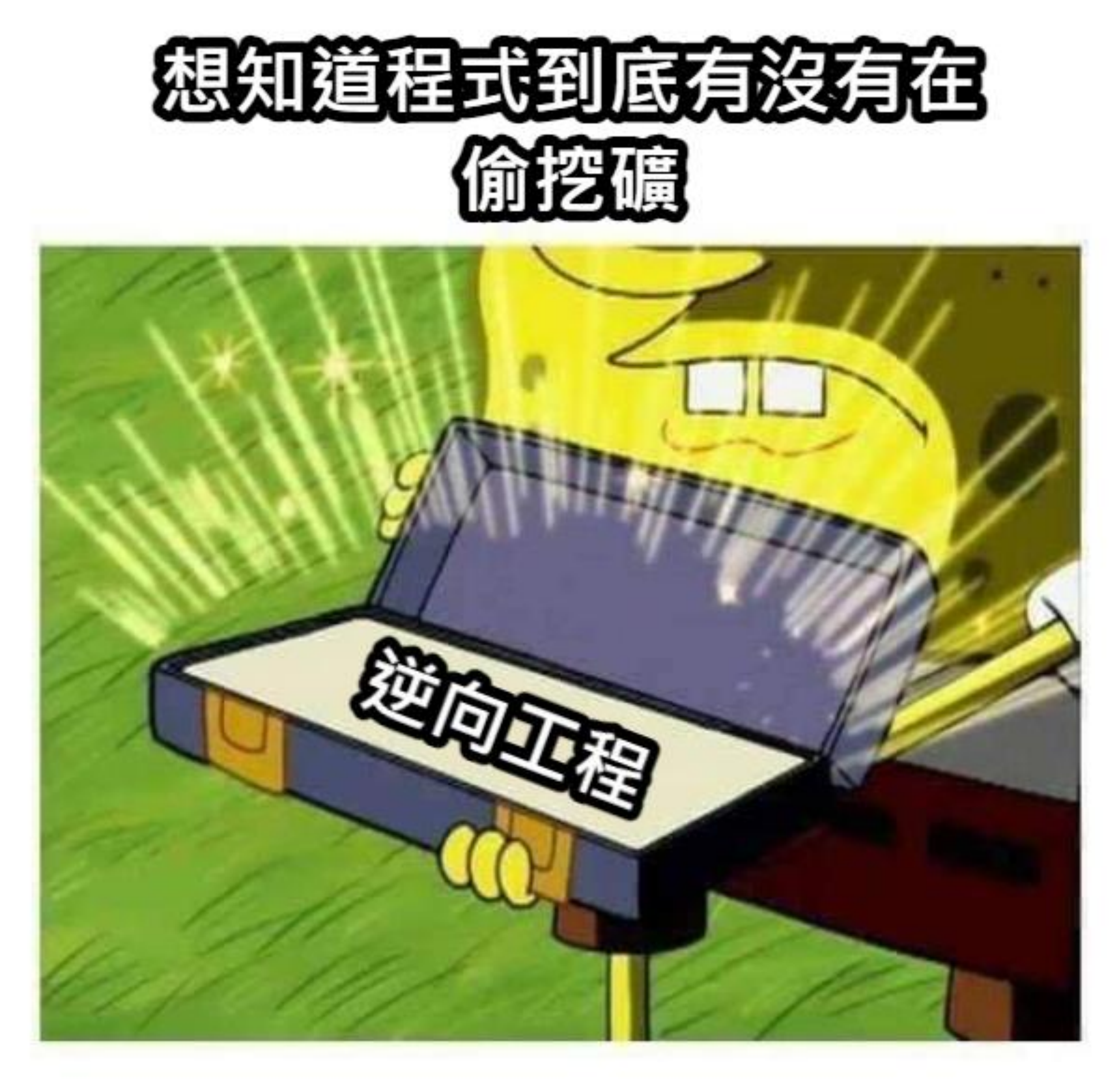
# whoami

- LJP / LJP-TW
- SQLab @ NYCU 碩一
- CTF @ 10sec / TSJ
- Pwner

Outline

- 逆向工程 What / Why / How
- x86
- Tools
- Calling Convention
- C -> x86
- Stack Frame
- Struct
- Endian
- Where to start?
- Compiler Optimization
- ASLR

逆向工程 What / Why / How


想知道程式到底有沒有在
偷挖礦
逆向工程

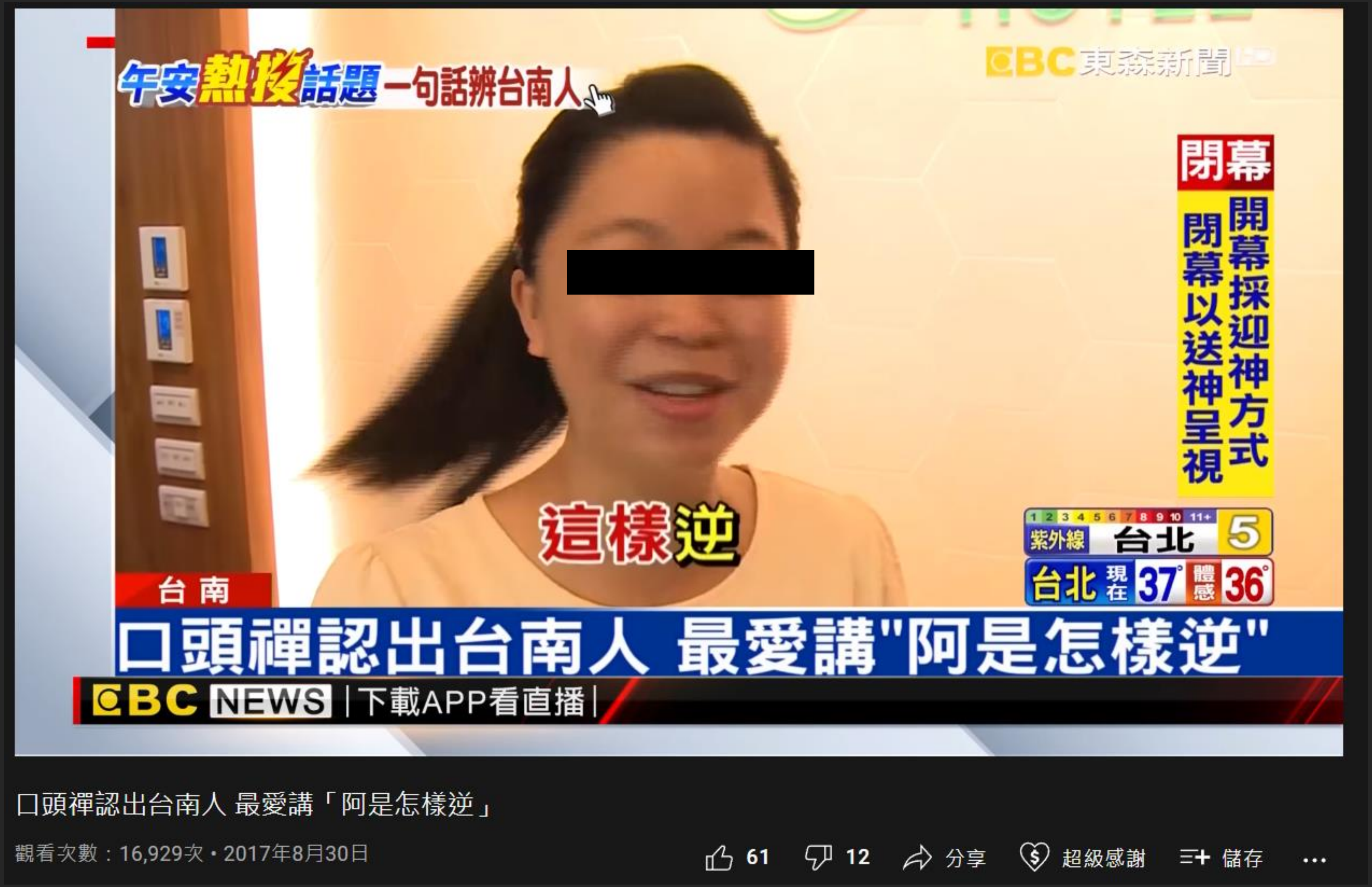
逆向工程 What / Why

- What:
  - 順向工程: 把想法變成 code, 再把 code 變成程式
  - 逆向工程: 用一些手段把程式變回 code, 再看懂作者的想法
  - ~~這樣單純看 code 算不算逆向 (?~~

- Why:
  - 沒有 source code 還想知道程式在做什麼

- 阿是怎樣逆

逆向工程 How



口頭禪認出台南人 最愛講「阿是怎樣逆」

觀看次數：16,929次・2017年8月30日

逆向工程 How

- 靜態分析
- 不把程式跑起來, 解析程式檔案
- 反組譯/反編譯, 使用人腦進行 debug
- IDA、Ghidra、PE-Bear、readelf、objdump

逆向工程 How

- 動態分析
- 把程式跑起來，觀察他的行為
- 在程式設定中斷點, 觀察程式暫存器/記憶體/…
- 用工具紀錄程式行為 e.g. 開檔案/網路連線/…
- Windbg preview、x64dbg、gdb

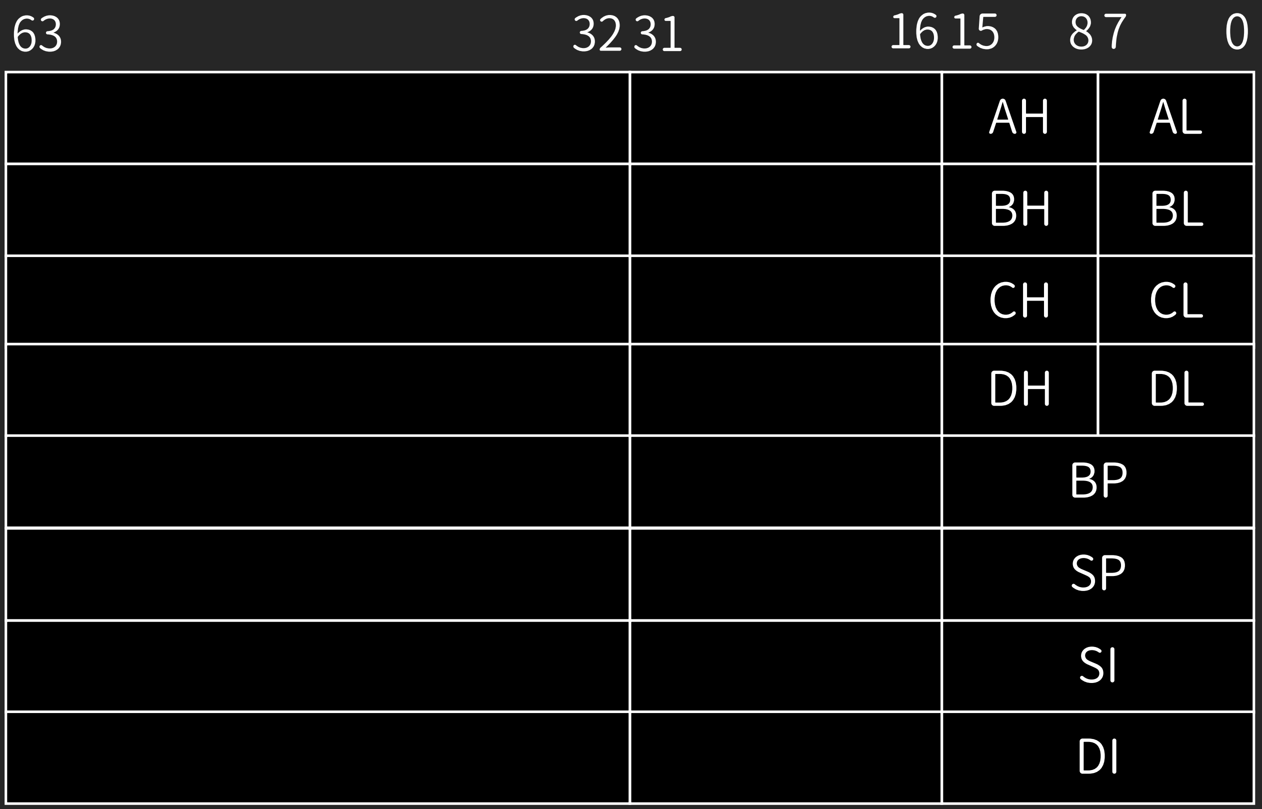
本堂課的設定…

- 程式底層運作原理不同, 逆向手段/需要的工具也不同
  - Java
  - .NET (e.g. C#, C++/CLI)
- CPU 指令集不同就差更多了
- 本堂課主要講由 C / C++ 寫成的程式, 指令集為 x86
- 接下來講講 x86

x86

x86 暫存器

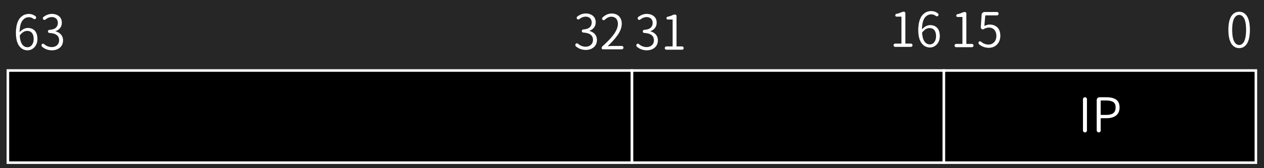
- 通用暫存器 (General-Purpose Registers)



16-bit	32-bit	64-bit	
AX	EAX	RAX	
BX	EBX	RBX	
CX	ECX	RCX	
DX	EDX	RDX	
BP	EBP	RBP	Base Pointer
SP	ESP	RSP	Stack Pointer
SI	ESI	RSI	
DI	EDI	RDI	

x86 暫存器

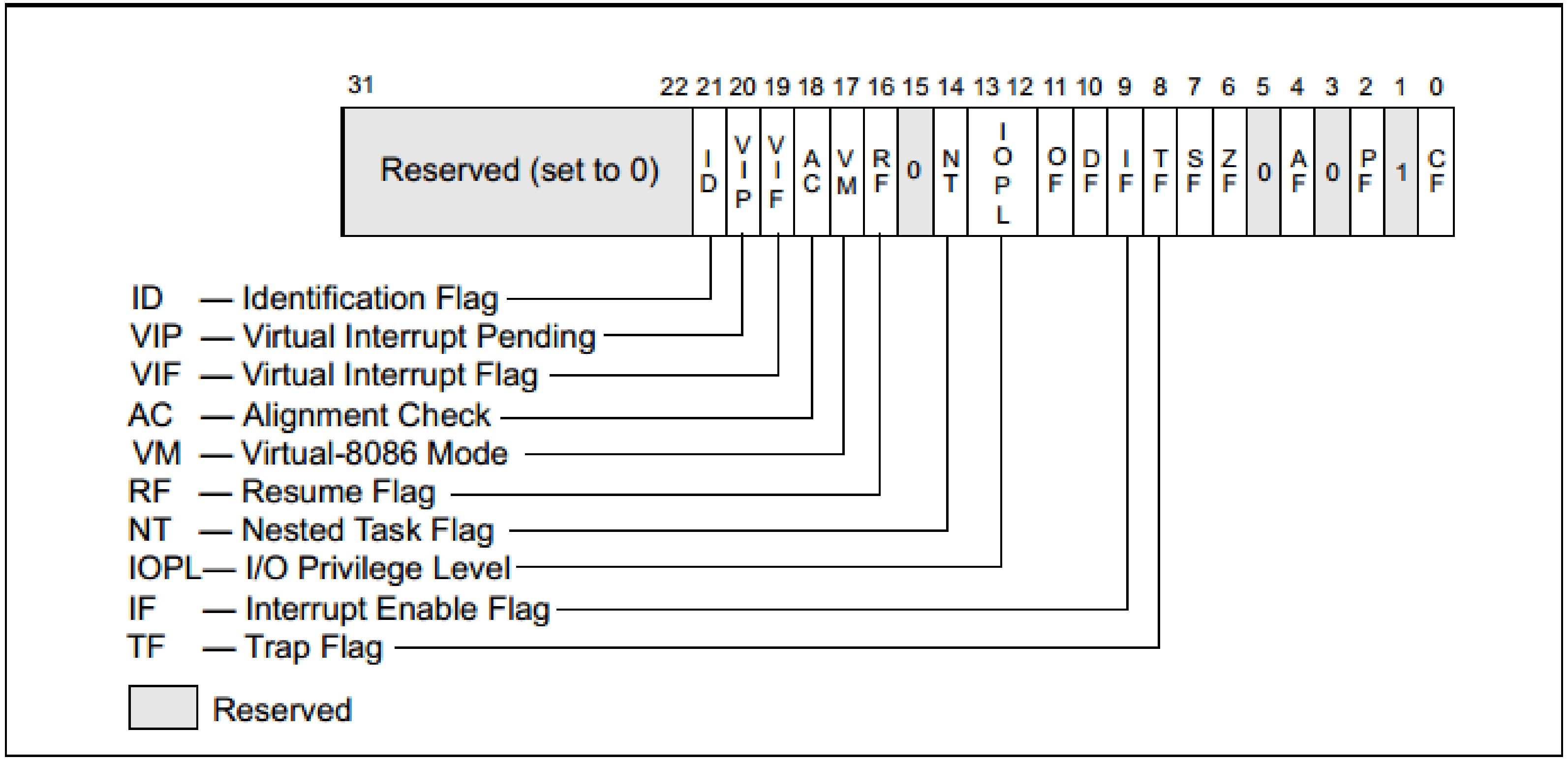
- 指令暫存器
  - Instruction Pointer Register
  - 或稱 Program Counter
- 存放下一條指令的位址



16-bit	32-bit	64-bit
IP	EIP	RIP

x86 暫存器

- EFLAGS



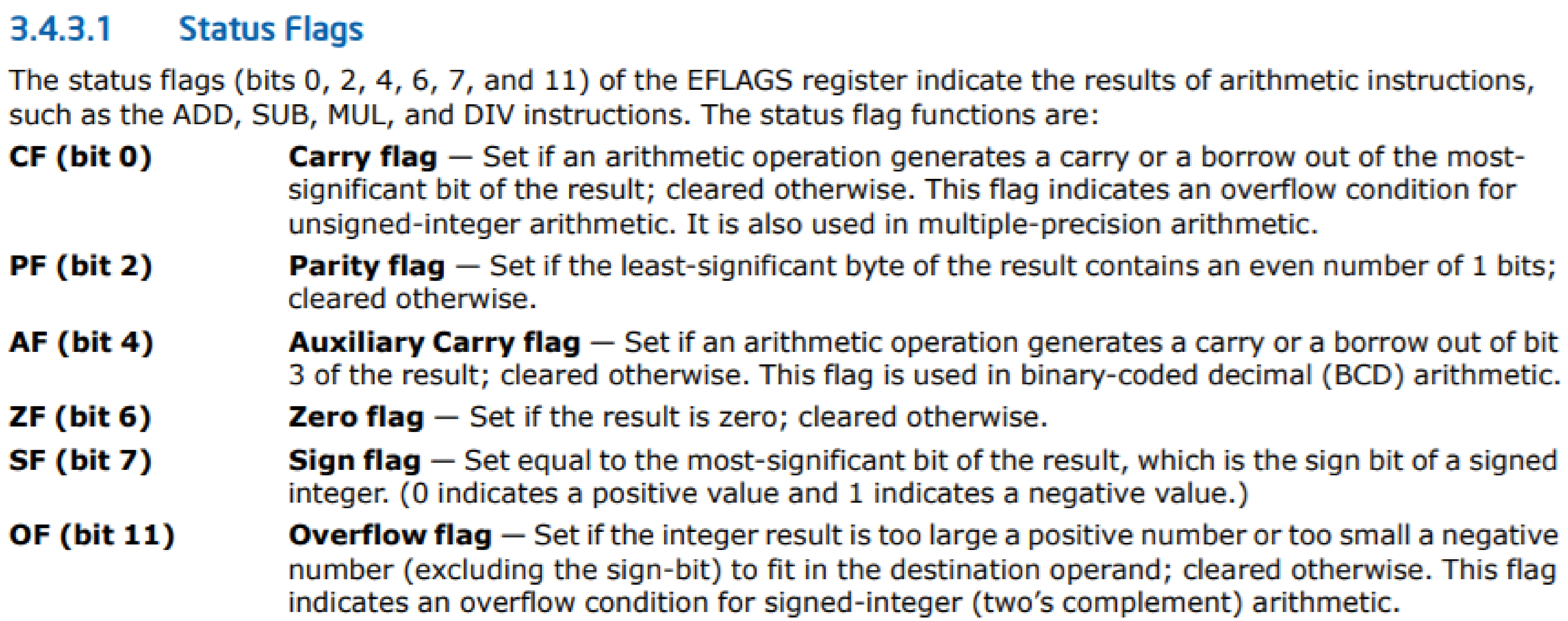
ID — Identification Flag
VIP — Virtual Interrupt Pending
VIF — Virtual Interrupt Flag
AC — Alignment Check
VM — Virtual-8086 Mode
RF — Resume Flag
NT — Nested Task Flag
IOPL— I/O Privilege Level
IF — Interrupt Enable Flag
TF — Trap Flag

▨ Reserved

Figure 2-4. System Flags in the EFLAGS Register

x86 暫存器

- EFLAGS



3.4.3.1 Status Flags

The status flags (bits 0, 2, 4, 6, 7, and 11) of the EFLAGS register indicate the results of arithmetic instructions, such as the ADD, SUB, MUL, and DIV instructions. The status flag functions are:

CF (bit 0) **Carry flag** — Set if an arithmetic operation generates a carry or a borrow out of the most-significant bit of the result; cleared otherwise. This flag indicates an overflow condition for unsigned-integer arithmetic. It is also used in multiple-precision arithmetic.

PF (bit 2) **Parity flag** — Set if the least-significant byte of the result contains an even number of 1 bits; cleared otherwise.

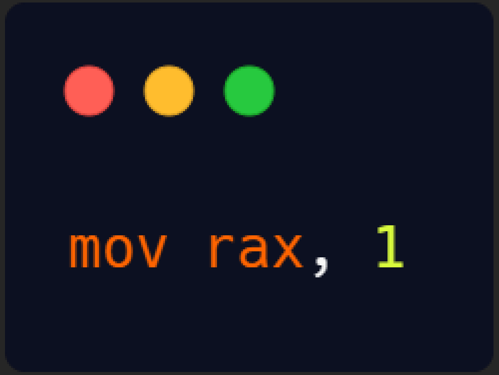
AF (bit 4) **Auxiliary Carry flag** — Set if an arithmetic operation generates a carry or a borrow out of bit 3 of the result; cleared otherwise. This flag is used in binary-coded decimal (BCD) arithmetic.

ZF (bit 6) **Zero flag** — Set if the result is zero; cleared otherwise.

SF (bit 7) **Sign flag** — Set equal to the most-significant bit of the result, which is the sign bit of a signed integer. (0 indicates a positive value and 1 indicates a negative value.)

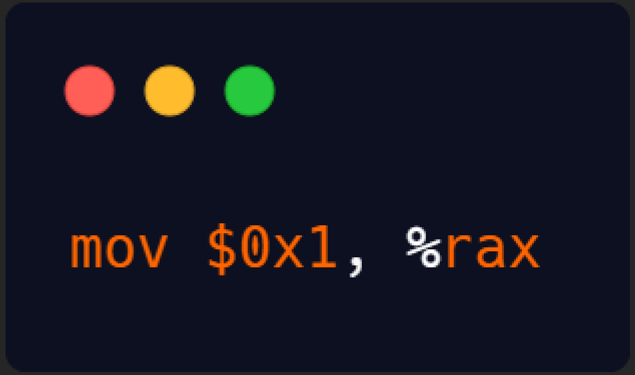
OF (bit 11) **Overflow flag** — Set if the integer result is too large a positive number or too small a negative number (excluding the sign-bit) to fit in the destination operand; cleared otherwise. This flag indicates an overflow condition for signed-integer (two's complement) arithmetic.

x86 指令



```
mov rax, 1
```

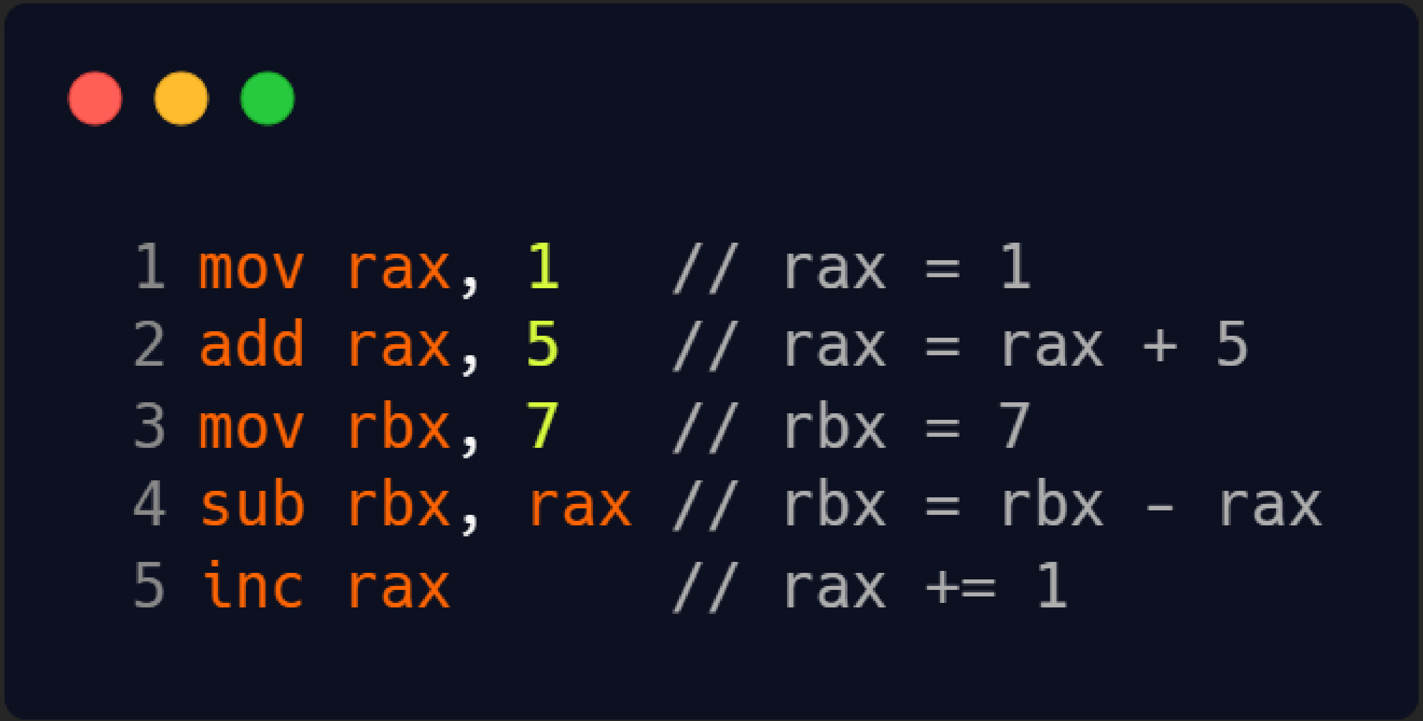
Intel syntax



```
mov $0x1, %rax
```

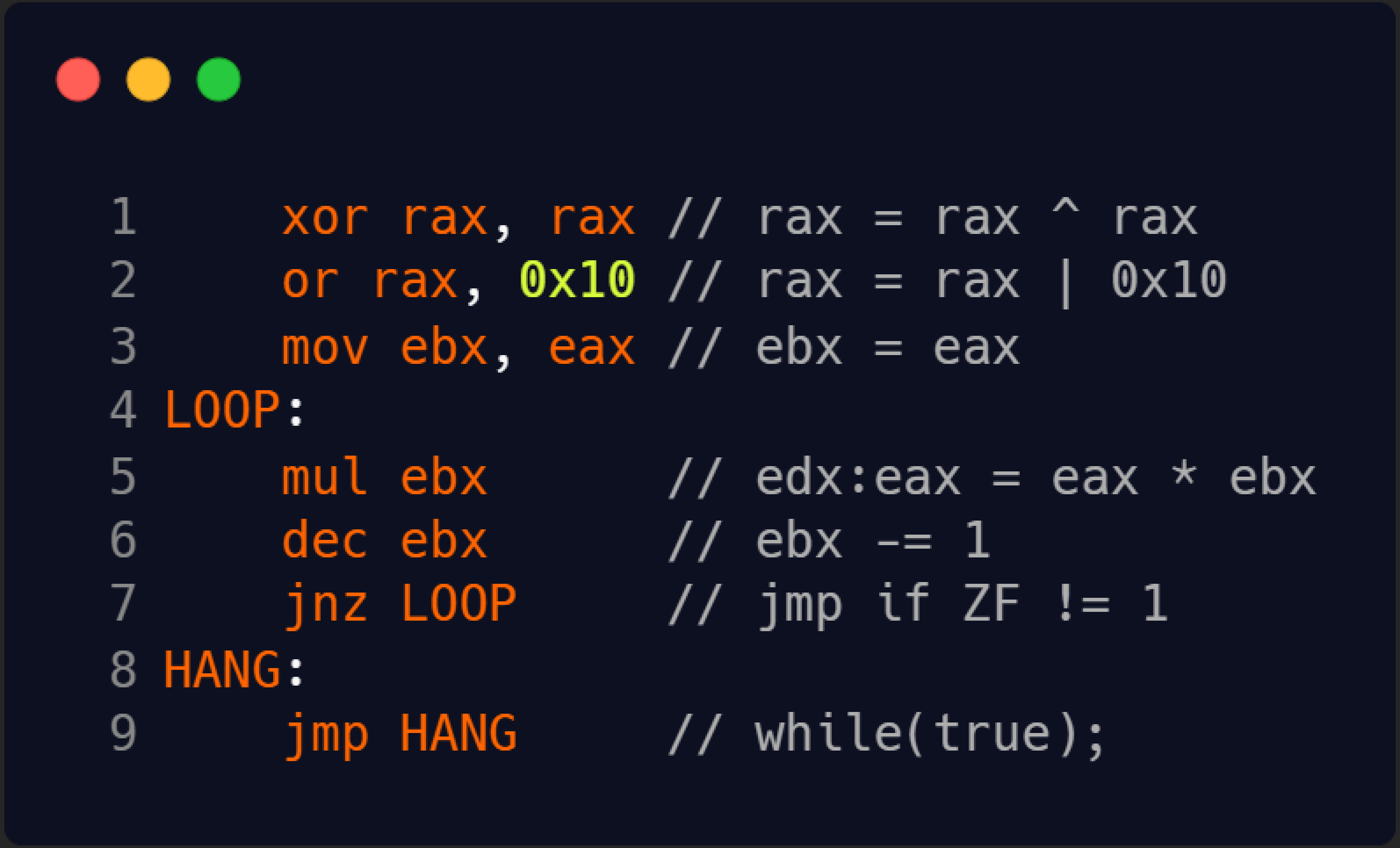
AT&T syntax

x86 指令



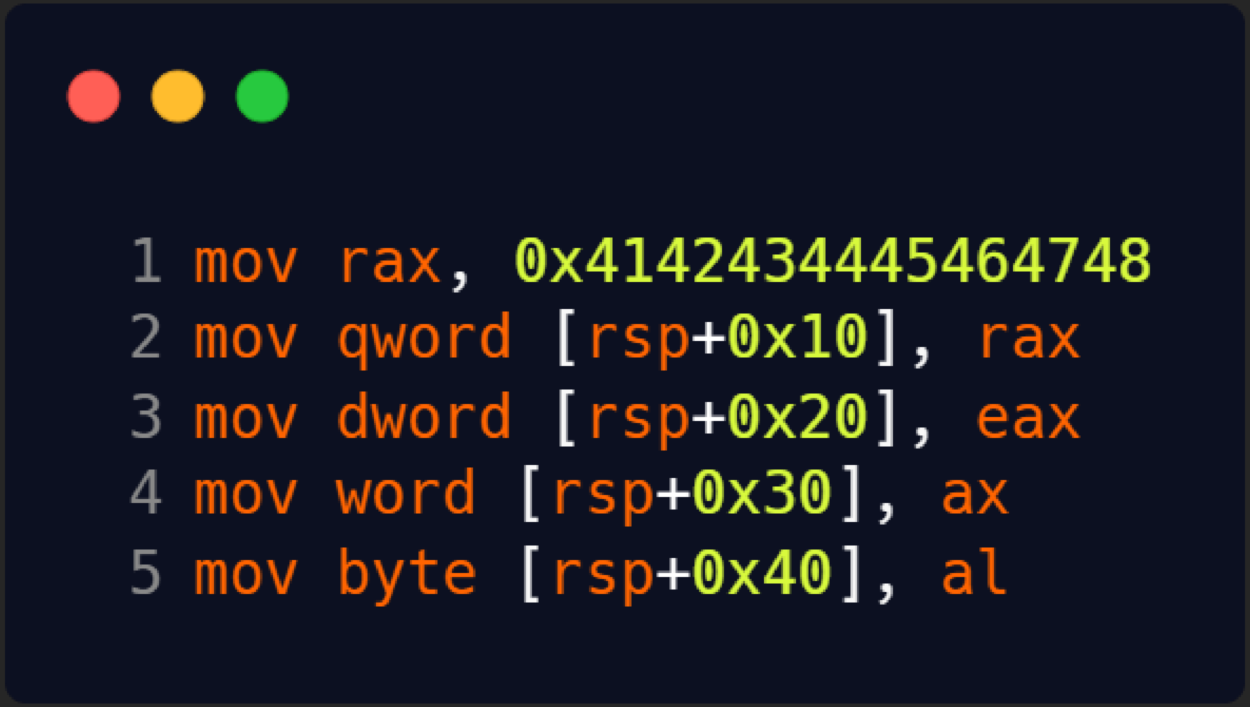
```
mov rax, 1   // rax = 1
add rax, 5   // rax = rax + 5
mov rbx, 7   // rbx = 7
sub rbx, rax // rbx = rbx - rax
inc rax      // rax += 1
```

x86 指令



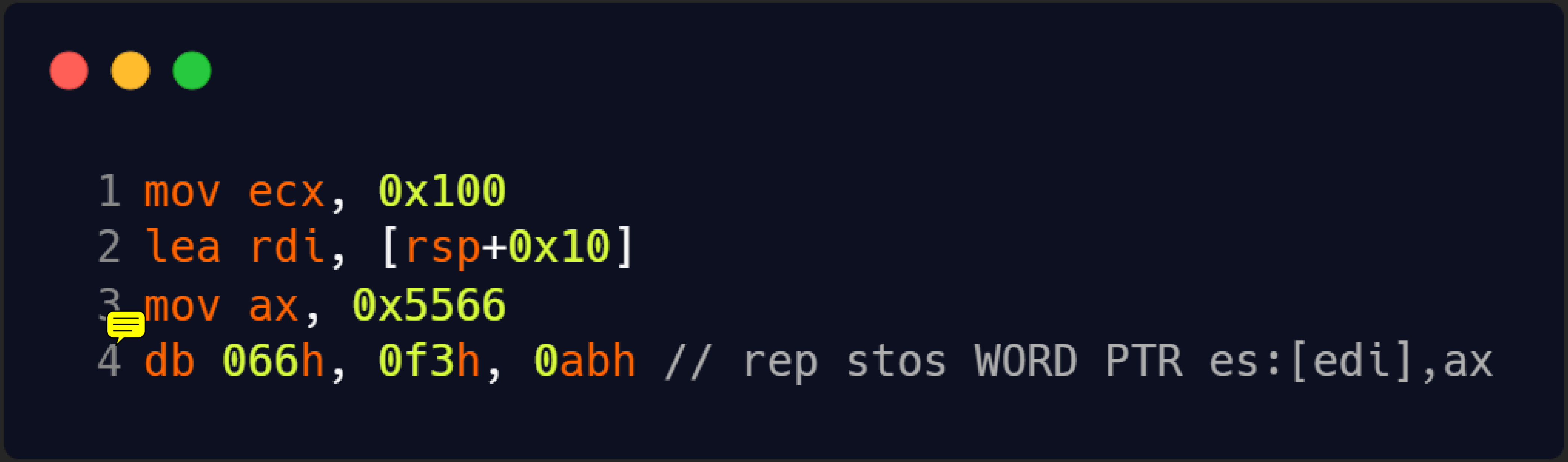
```
    xor rax, rax // rax = rax ^ rax
    or rax, 0x10 // rax = rax | 0x10
    mov ebx, eax // ebx = eax
LOOP:
    mul ebx      // edx:eax = eax * ebx
    dec ebx      // ebx -= 1
    jnz LOOP     // jmp if ZF != 1
HANG:
    jmp HANG     // while(true);
```

x86 指令



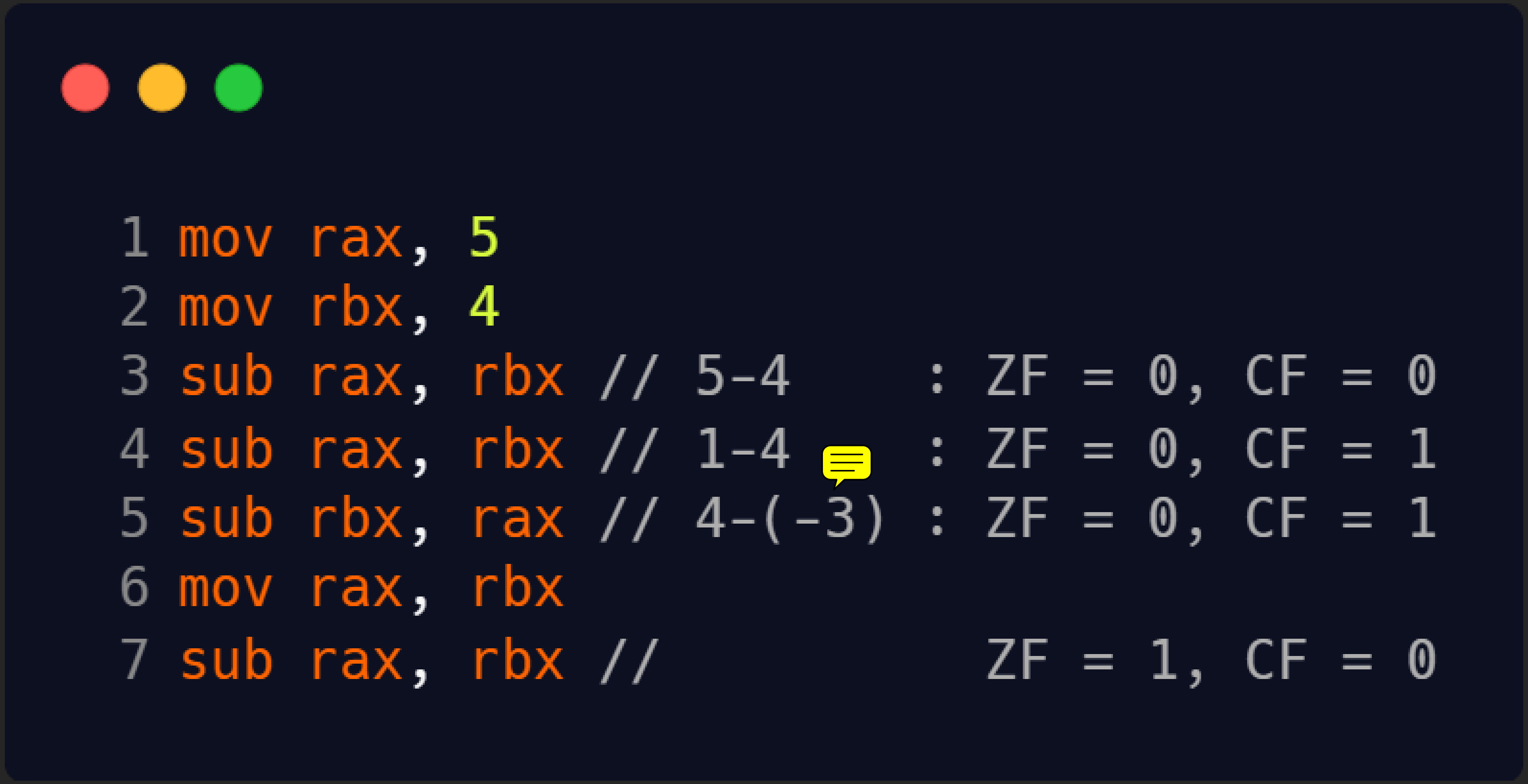
```
mov rax, 0x4142434445464748
mov qword [rsp+0x10], rax
mov dword [rsp+0x20], eax
mov word [rsp+0x30], ax
mov byte [rsp+0x40], al
```

x86 指令



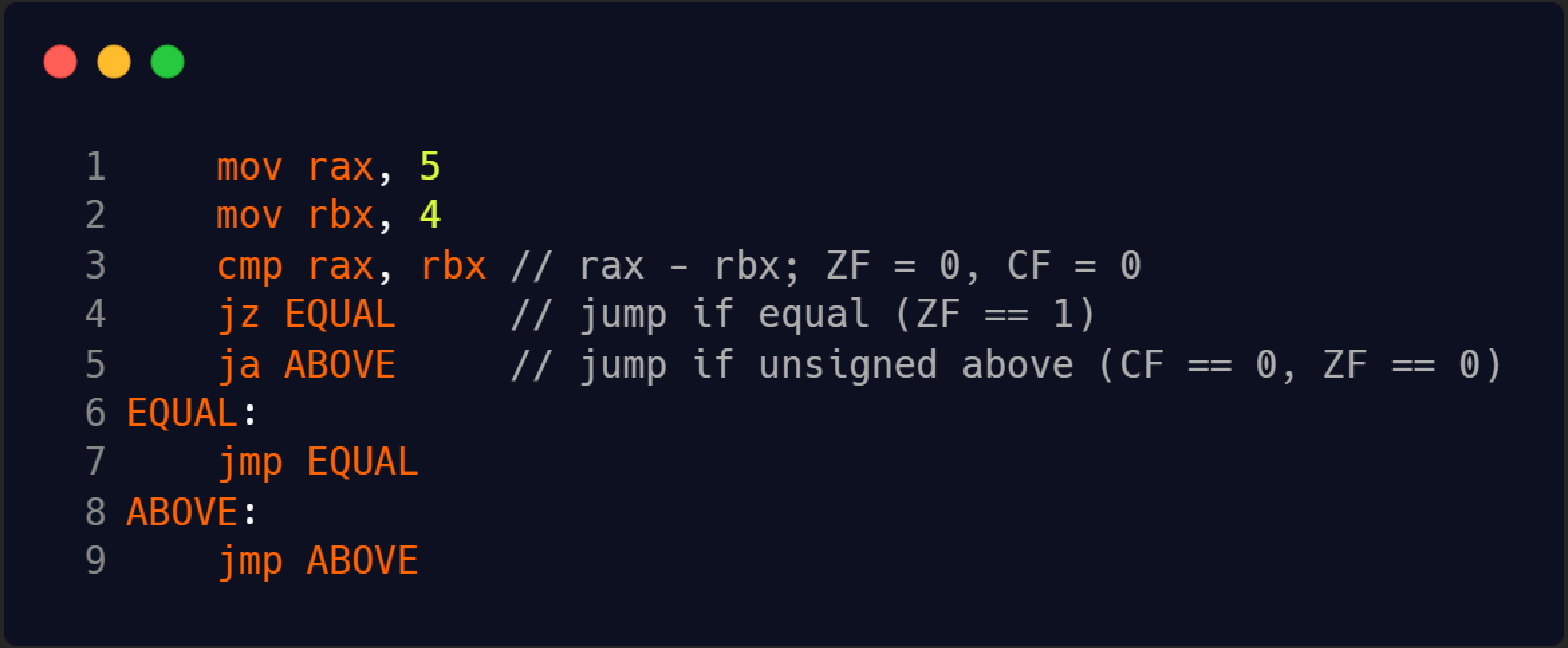
```
mov ecx, 0x100
lea rdi, [rsp+0x10]
mov ax, 0x5566
db 066h, 0f3h, 0abh // rep stos WORD PTR es:[edi],ax
```

x86 指令



```
mov rax, 5
mov rbx, 4
sub rax, rbx // 5-4    : ZF = 0, CF = 0
sub rax, rbx // 1-4    : ZF = 0, CF = 1
sub rbx, rax // 4-(-3) : ZF = 0, CF = 1
mov rax, rbx
sub rax, rbx //          ZF = 1, CF = 0
```

x86 指令



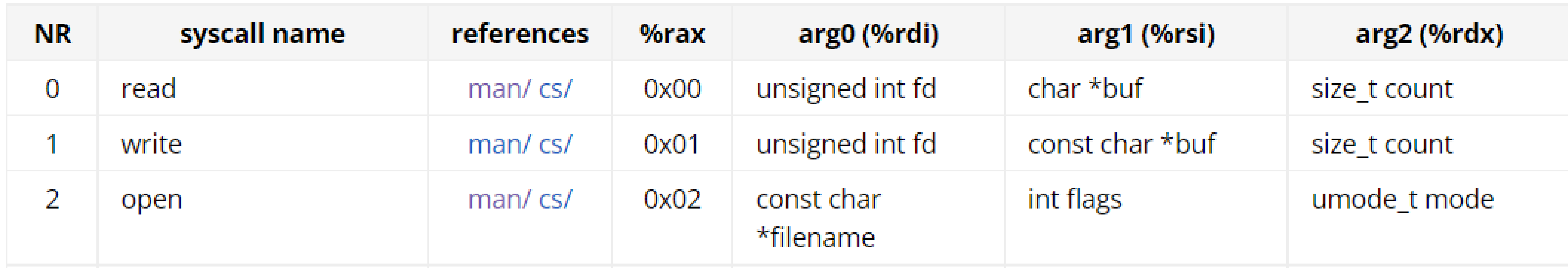
```
    mov rax, 5
    mov rbx, 4
    cmp rax, rbx // rax - rbx; ZF = 0, CF = 0
    jz EQUAL     // jump if equal (ZF == 1)
    ja ABOVE     // jump if unsigned above (CF == 0, ZF == 0)
EQUAL:
    jmp EQUAL
ABOVE:
    jmp ABOVE
```

Jump quick ref: http://unixwiz.net/techtips/x86-jumps.html

x86 指令

Linux System Call

- rax syscall (rdi, rsi, rdx, r10, r8, r9)



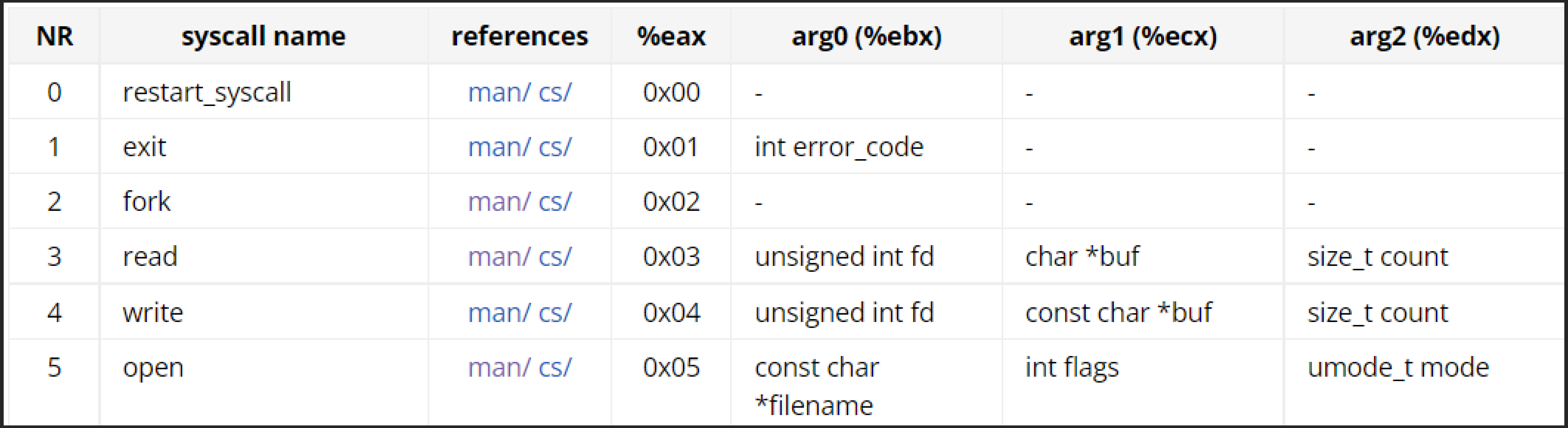
NR	syscall name	references	%rax	arg0 (%rdi)	arg1 (%rsi)	arg2 (%rdx)
0	read	man/ cs/	0x00	unsigned int fd	char *buf	size_t count
1	write	man/ cs/	0x01	unsigned int fd	const char *buf	size_t count
2	open	man/ cs/	0x02	const char *filename	int flags	umode_t mode

Ref: https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#x86_64-64_bit

x86 指令

Linux System Call

- eax int 0x80 (ebx, ecx, edx, esi, edi, ebp)

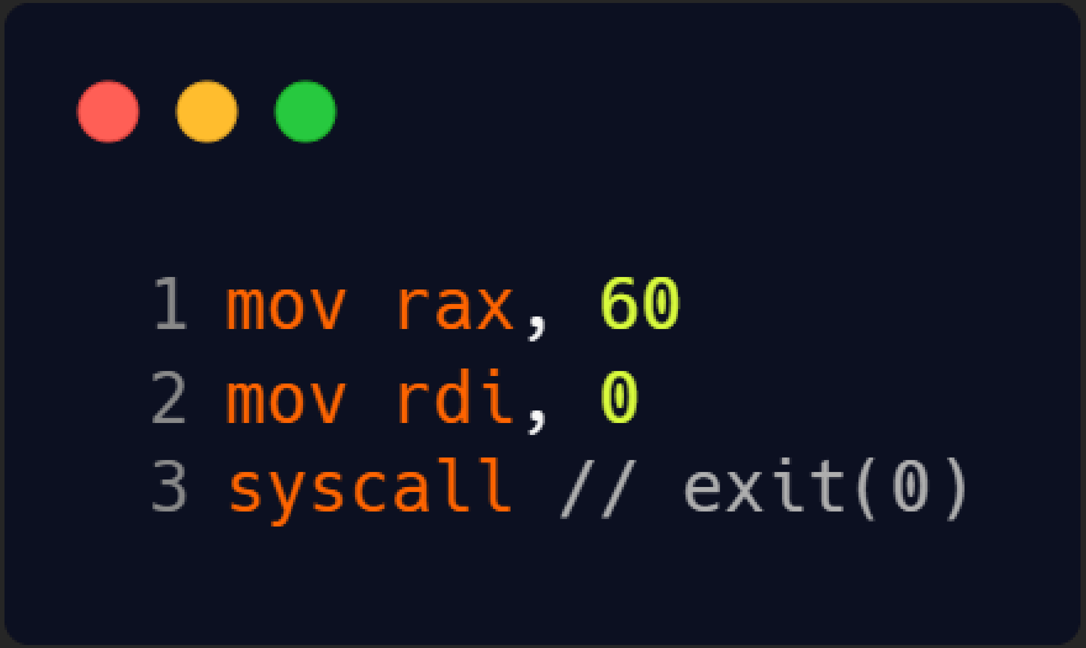


NR	syscall name	references	%eax	arg0 (%ebx)	arg1 (%ecx)	arg2 (%edx)
0	restart_syscall	man/ cs/	0x00	-	-	-
1	exit	man/ cs/	0x01	int error_code	-	-
2	fork	man/ cs/	0x02	-	-	-
3	read	man/ cs/	0x03	unsigned int fd	char *buf	size_t count
4	write	man/ cs/	0x04	unsigned int fd	const char *buf	size_t count
5	open	man/ cs/	0x05	const char *filename	int flags	umode_t mode

Ref: https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#x86-32_bit

x86 指令

Linux System Call



```
mov rax, 60
mov rdi, 0
syscall // exit(0)
```

Ref: https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#x86_64-64_bit

x86 指令

- 遇到沒看過的就菇狗關鍵字 “x86 <指令>”

Tools

IDA / gdb

Tools - IDA

- 靜態分析工具
  - 反組譯
  - 反編譯
  - Cross References (Xrefs)
  - 函數 / 變數改名
  - 註解
  - 定義 struct
  - 內建 python API 可以通過腳本做事
  - 各種 plug-in

Tools - IDA

- 常用快捷鍵
  - Space: 在 Text View / Graph View 切換
  - Tab: 在視窗之間切換
  - ; / Insert: 註解
  - x: 秀出 Xrefs
  - n: 改名
  - y: 改型別
  - h: 改表示方式 (dec / hex)
  - u: 取消定義
  - a: 當成字串

DEMO

Tools - gdb

- 動態分析工具
  - 設定中斷點
  - 執行程式
  - 查看記憶體 / 暫存器
  - 查看 address space

Tools - gdb

- 常用語法
  - b: 設定中斷點
  - r: 執行程式
  - c: 繼續執行
  - si: 步入指令
  - ni: 步過指令
  - x: 顯示記憶體內容
  - vmmap: 查看 address space

DEMO

Tools

- 工具主要就分成這兩類
  - 靜態分析工具
  - 動態分析工具

- 同一類工具的功能大同小異, 順手最重要

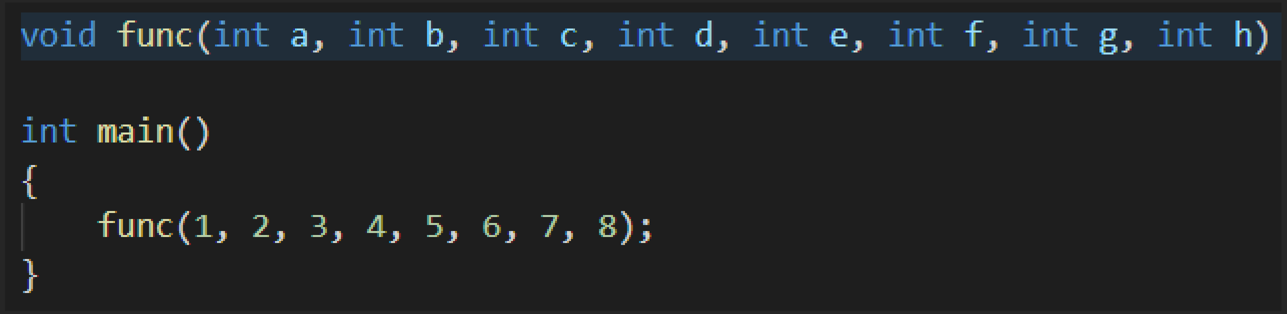
Lab 1

Calling Convention

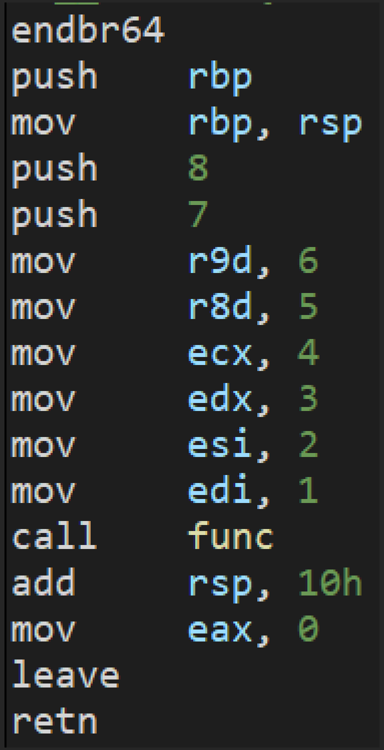
Calling Convention

- 約定了呼叫函數時如何傳遞參數
- x64
  - Windows
    - Function(rcx, rdx, r8, r9)
  - Linux
    - Function(rdi, rsi, rdx, rcx, r8, r9)
  - 多的放 stack
- x32
  - 都放 stack

Calling Convention



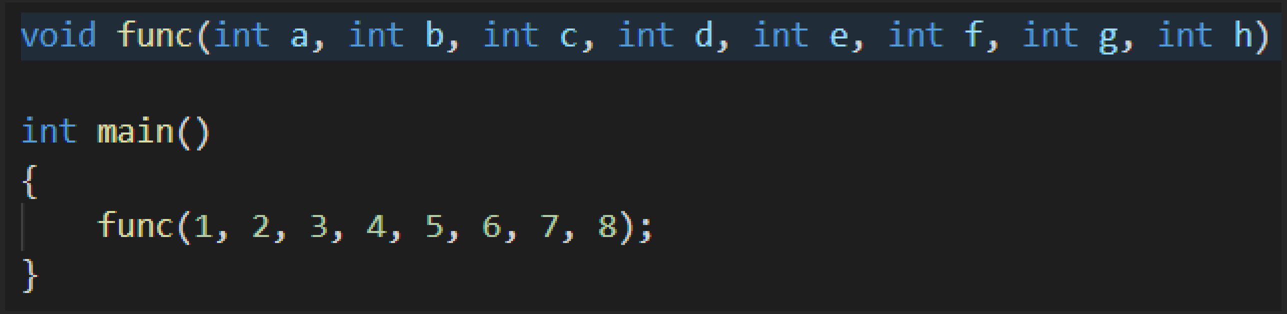
```c
void func(int a, int b, int c, int d, int e, int f, int g, int h)

int main()
{
    func(1, 2, 3, 4, 5, 6, 7, 8);
}
```



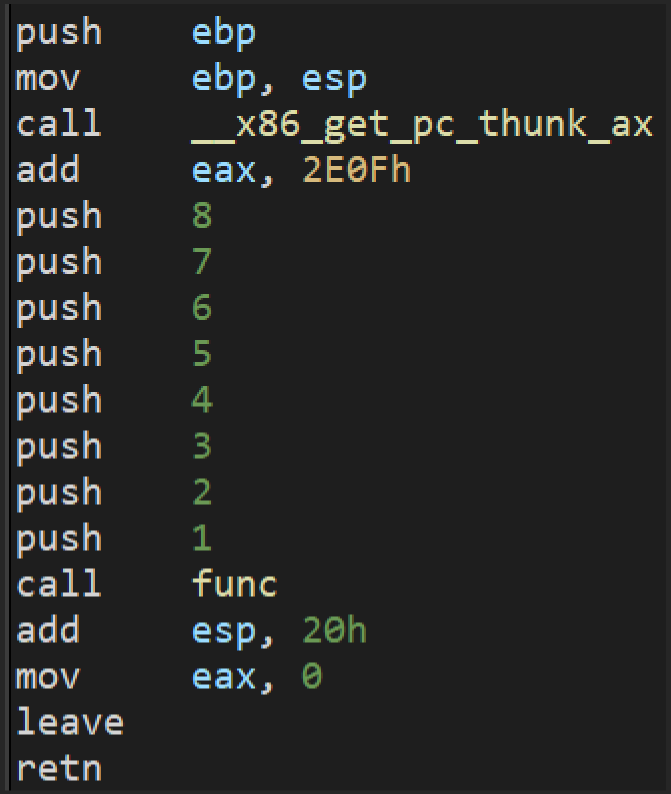
```
endbr64
push    rbp
mov     rbp, rsp
push    8
push    7
mov     r9d, 6
mov     r8d, 5
mov     ecx, 4
mov     edx, 3
mov     esi, 2
mov     edi, 1
call    func
add     rsp, 10h
mov     eax, 0
leave
retn
```

x86-64

Calling Convention



```
void func(int a, int b, int c, int d, int e, int f, int g, int h)

int main()
{
    func(1, 2, 3, 4, 5, 6, 7, 8);
}
```



```
push    ebp
mov     ebp, esp
call    __x86_get_pc_thunk_ax
add     eax, 2E0Fh
push    8
push    7
push    6
push    5
push    4
push    3
push    2
push    1
call    func
add     esp, 20h
mov     eax, 0
leave
retn
```

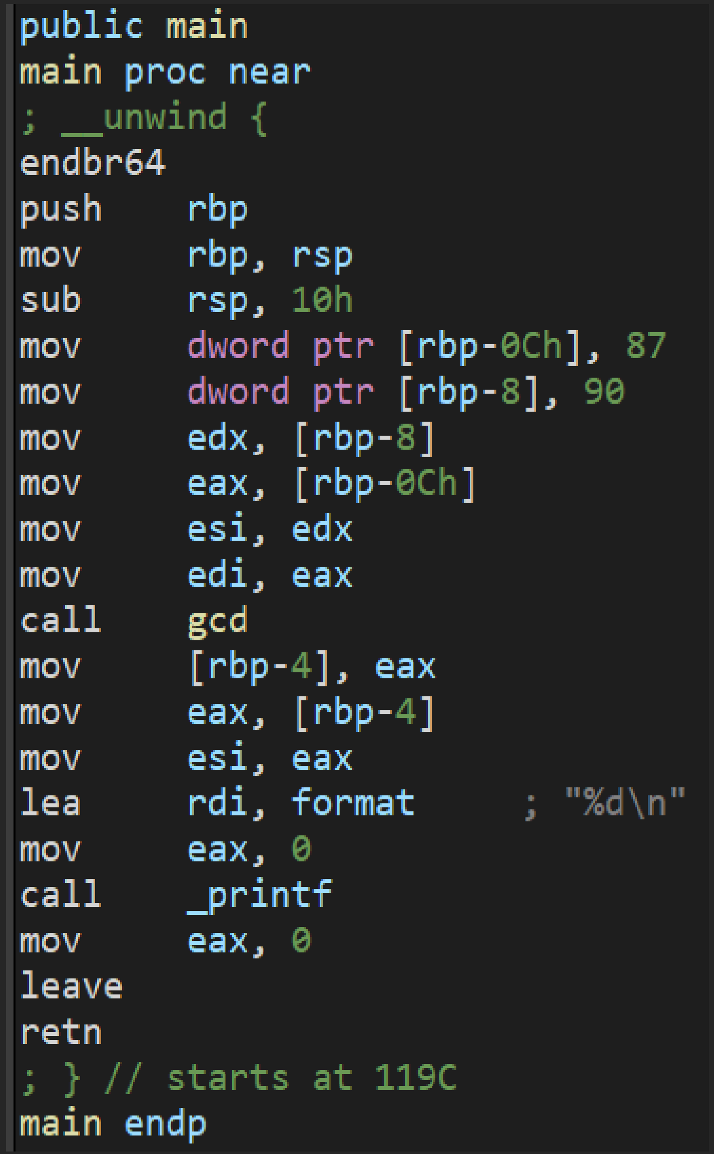
x86

C -> x86

C -> x86

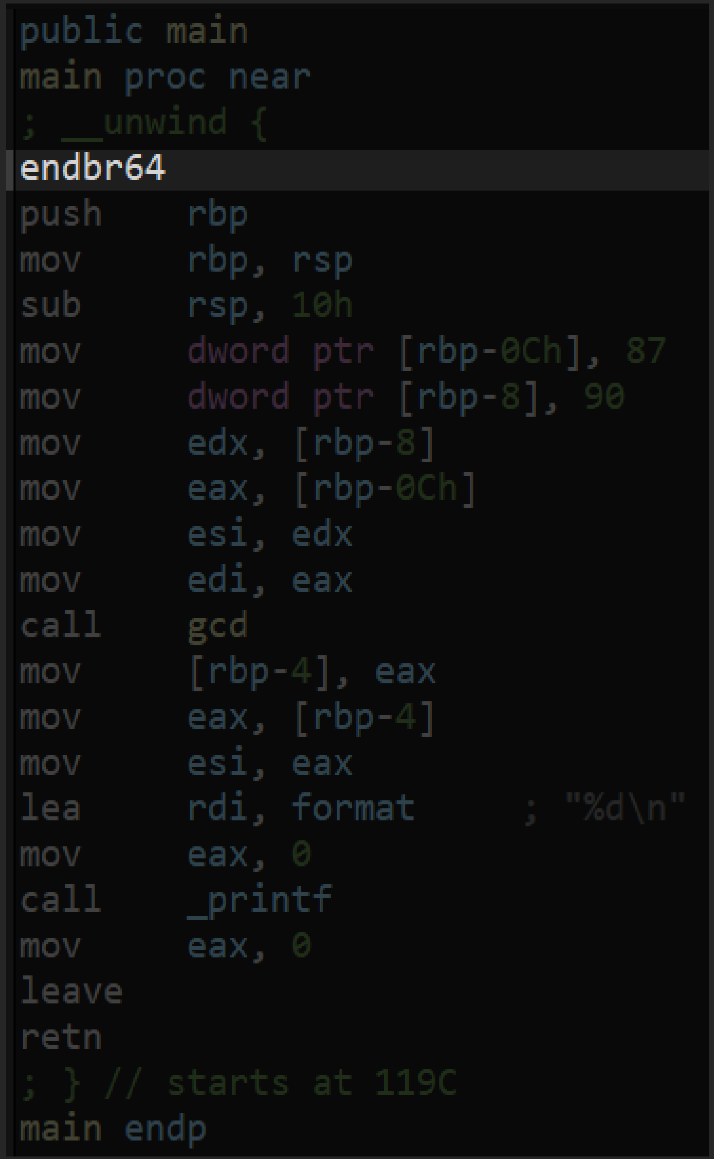
- 前面的程式都是手寫的組語
- 這個章節來看一下 C 編出來的組語

C -> x86



```
public main
main proc near
; __unwind {
endbr64
push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     dword ptr [rbp-0Ch], 87
mov     dword ptr [rbp-8], 90
mov     edx, [rbp-8]
mov     eax, [rbp-0Ch]
mov     esi, edx
mov     edi, eax
call    gcd
mov     [rbp-4], eax
mov     eax, [rbp-4]
mov     esi, eax
lea     rdi, format     ; "%d\n"
mov     eax, 0
call    _printf
mov     eax, 0
leave
retn
; } // starts at 119C
main endp
```

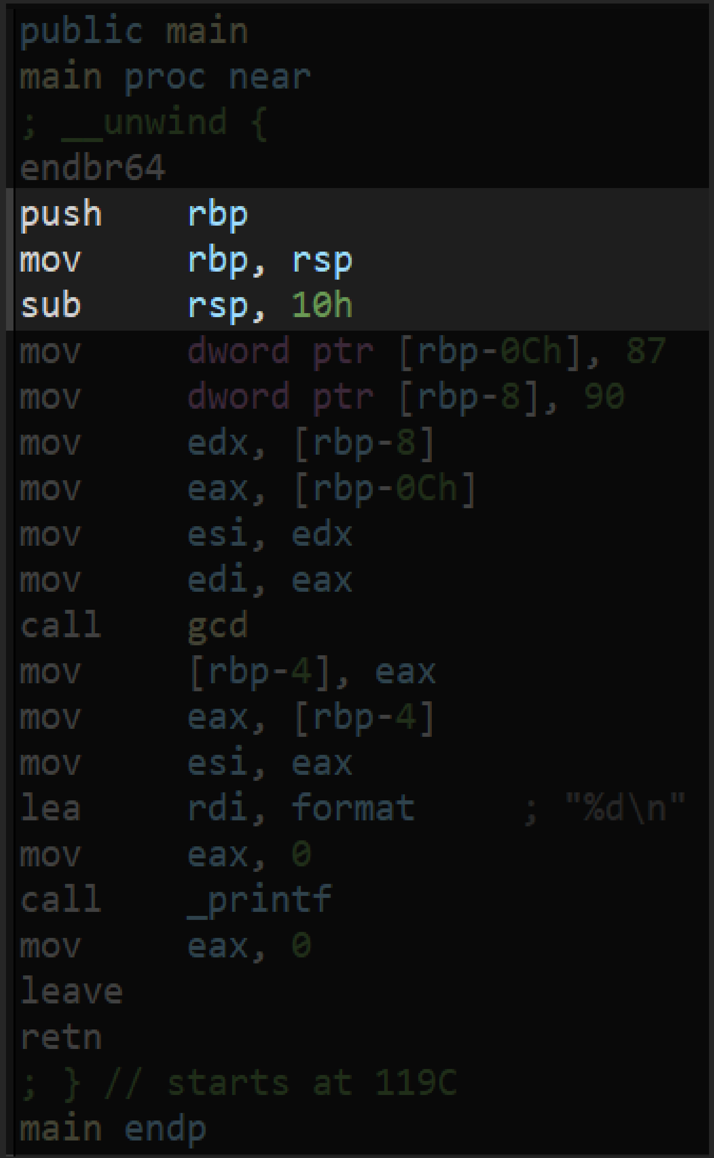
C -> x86



```
public main
main proc near
; __unwind {
endbr64
push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     dword ptr [rbp-0Ch], 87
mov     dword ptr [rbp-8], 90
mov     edx, [rbp-8]
mov     eax, [rbp-0Ch]
mov     esi, edx
mov     edi, eax
call    gcd
mov     [rbp-4], eax
mov     eax, [rbp-4]
mov     esi, eax
lea     rdi, format     ; "%d\n"
mov     eax, 0
call    _printf
mov     eax, 0
leave
retn
; } // starts at 119C
main endp
```

For security
無運算意義

C -> x86

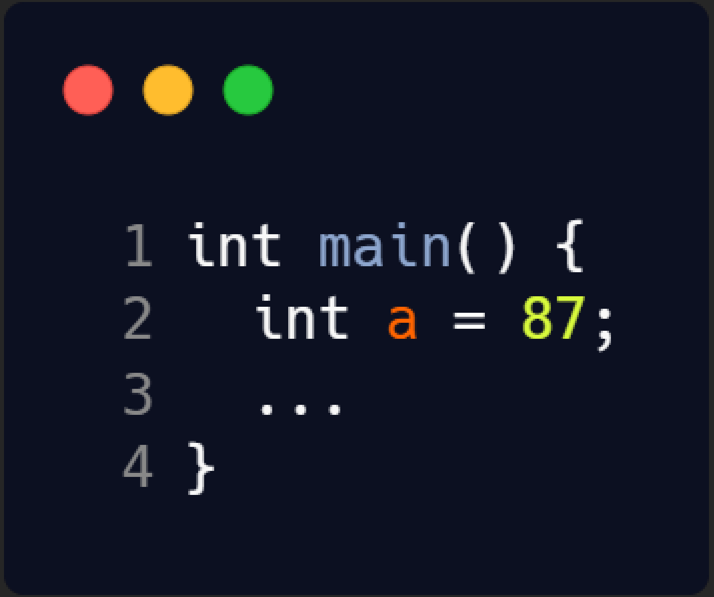


```
public main
main proc near
; __unwind {
endbr64
push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     dword ptr [rbp-0Ch], 87
mov     dword ptr [rbp-8], 90
mov     edx, [rbp-8]
mov     eax, [rbp-0Ch]
mov     esi, edx
mov     edi, eax
call    gcd
mov     [rbp-4], eax
mov     eax, [rbp-4]
mov     esi, eax
lea     rdi, format     ; "%d\n"
mov     eax, 0
call    _printf
mov     eax, 0
leave
retn
; } // starts at 119C
main endp
```

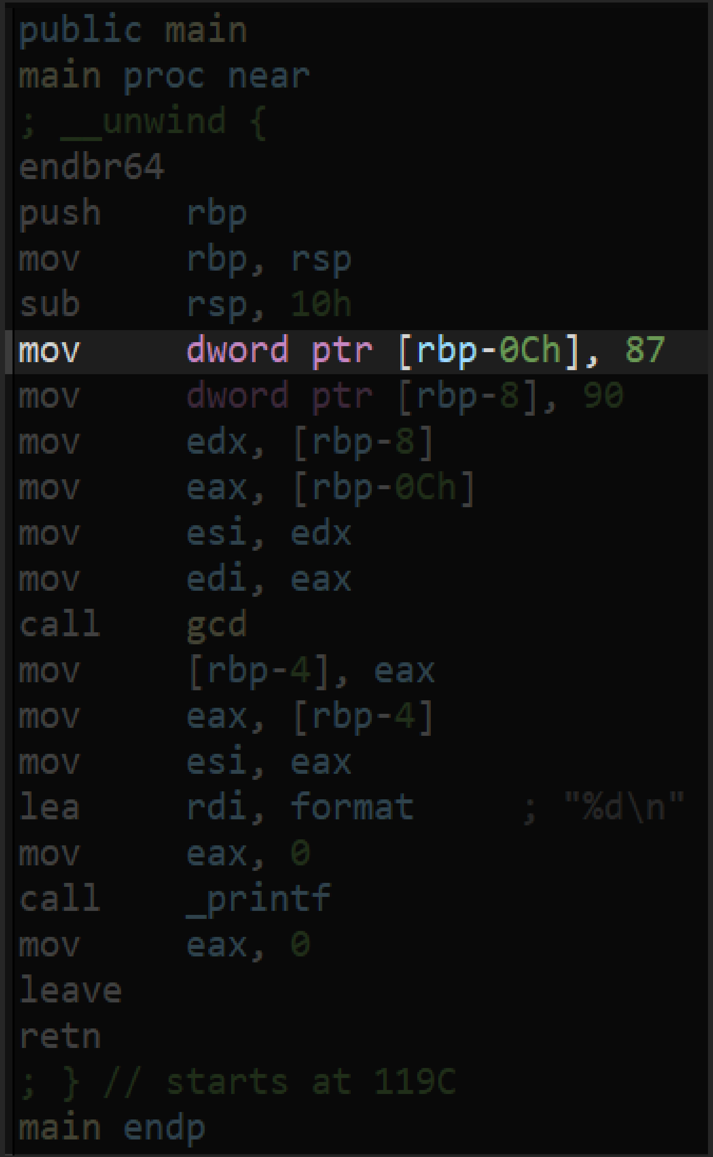
For security
無運算意義

Stack Frame
Prologue
(下個章節講)

C -> x86



```c
int main() {
  int a = 87;
  ...
}
```

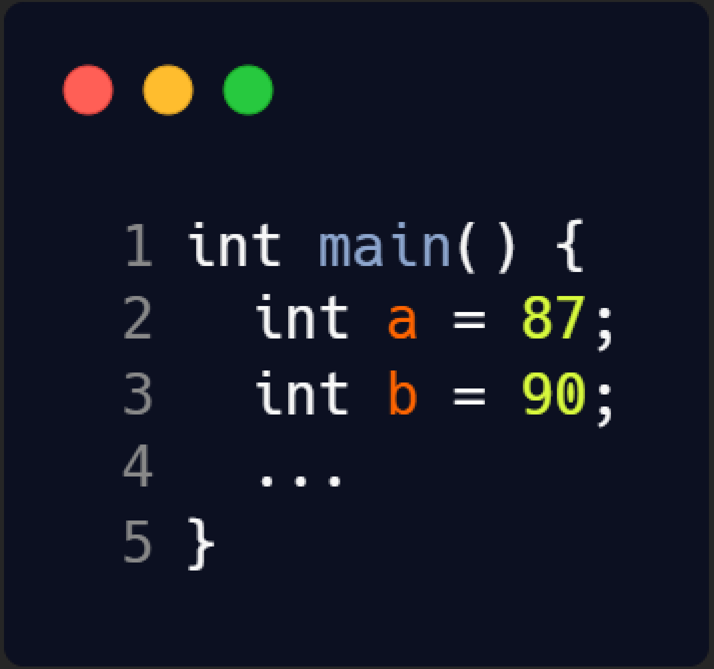


```asm
public main
main proc near
; __unwind {
endbr64
push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     dword ptr [rbp-0Ch], 87
mov     dword ptr [rbp-8], 90
mov     edx, [rbp-8]
mov     eax, [rbp-0Ch]
mov     esi, edx
mov     edi, eax
call    gcd
mov     [rbp-4], eax
mov     eax, [rbp-4]
mov     esi, eax
lea     rdi, format     ; "%d\n"
mov     eax, 0
call    _printf
mov     eax, 0
leave
retn
; } // starts at 119C
main endp
```

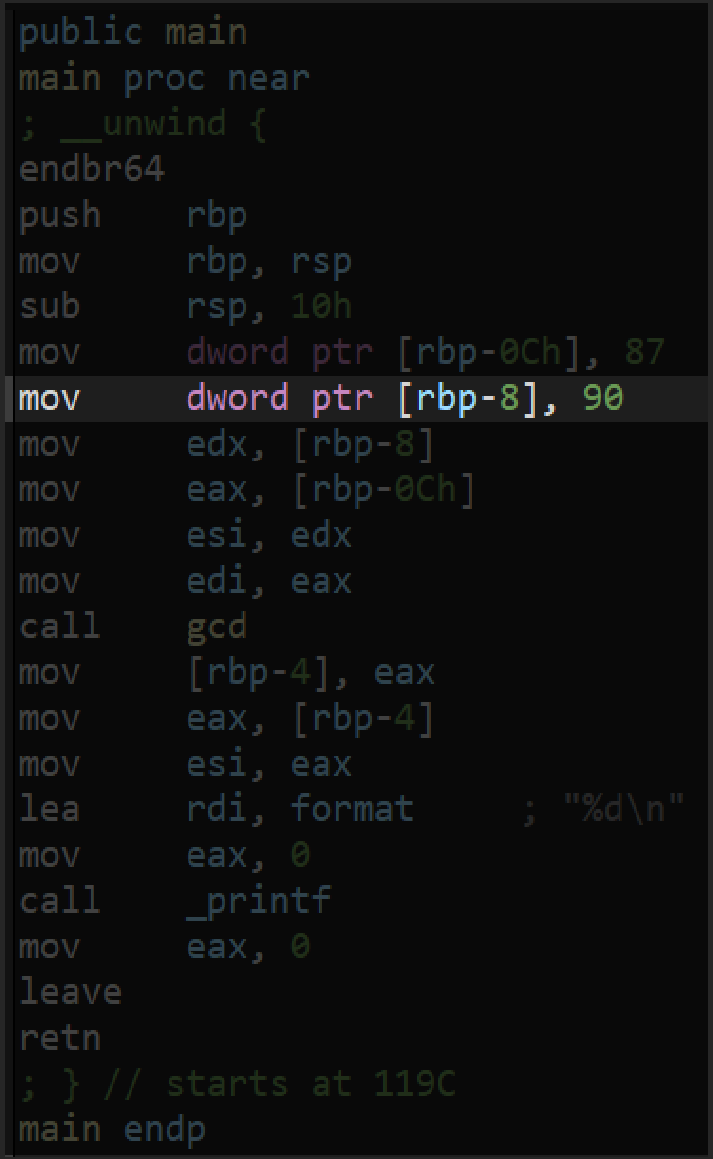
For security
無運算意義

Stack Frame
Prologue
(下個章節講)

C -> x86



```c
int main() {
   int a = 87;
   int b = 90;
   ...
}
```

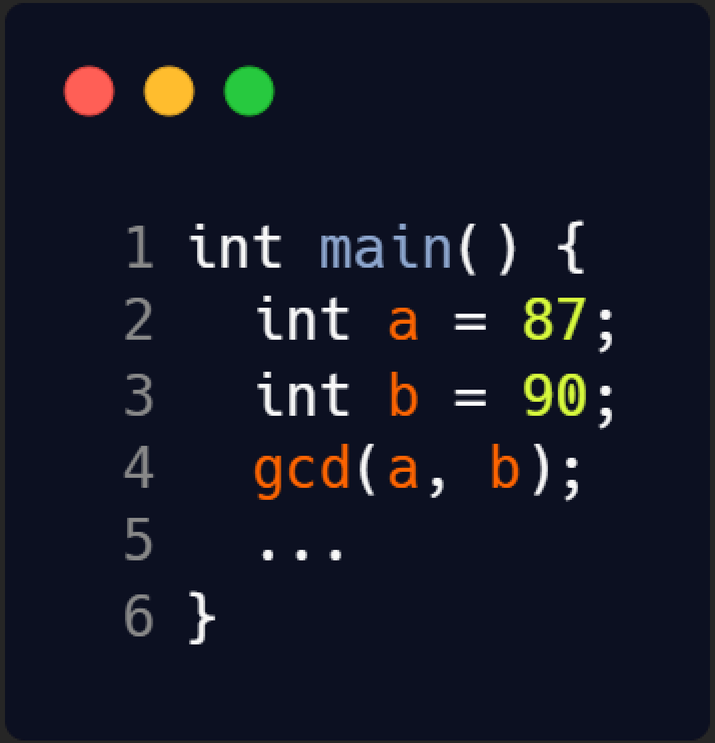


```
public main
main proc near
; __unwind {
endbr64
push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     dword ptr [rbp-0Ch], 87
mov     dword ptr [rbp-8], 90
mov     edx, [rbp-8]
mov     eax, [rbp-0Ch]
mov     esi, edx
mov     edi, eax
call    gcd
mov     [rbp-4], eax
mov     eax, [rbp-4]
mov     esi, eax
lea     rdi, format     ; "%d\n"
mov     eax, 0
call    _printf
mov     eax, 0
leave
retn
; } // starts at 119C
main endp
```

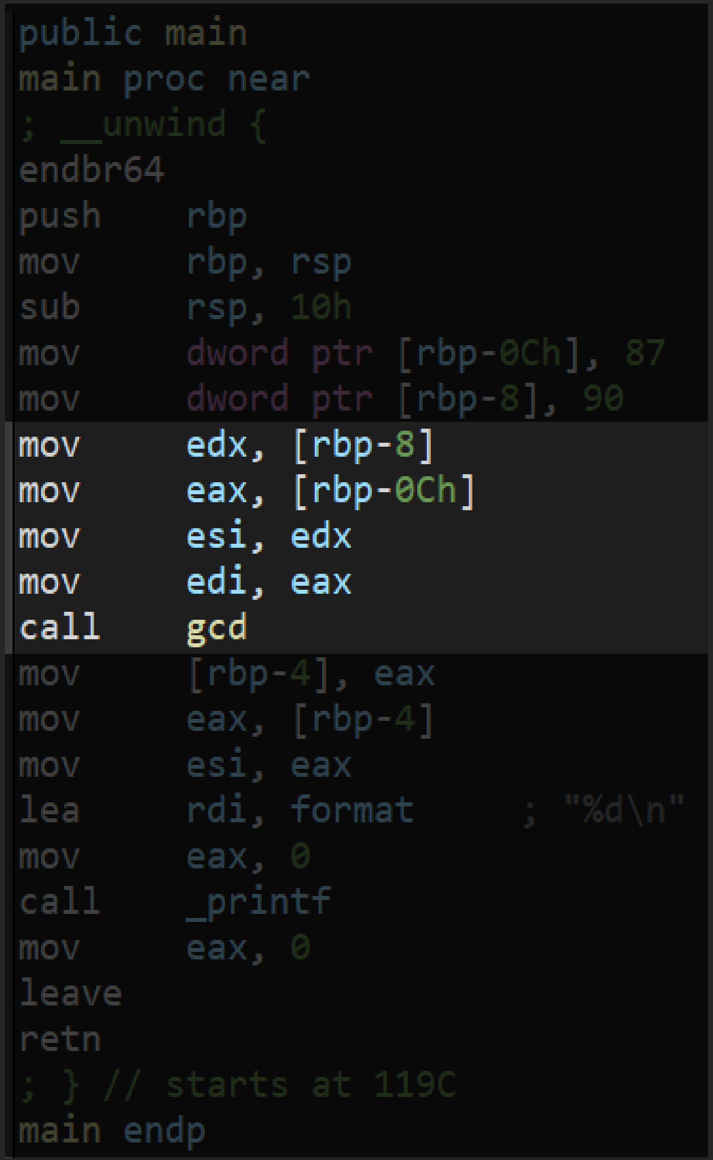
For security
無運算意義

Stack Frame
Prologue
(下個章節講)

C -> x86



```c
int main() {
  int a = 87;
  int b = 90;
  gcd(a, b);
  ...
}
```

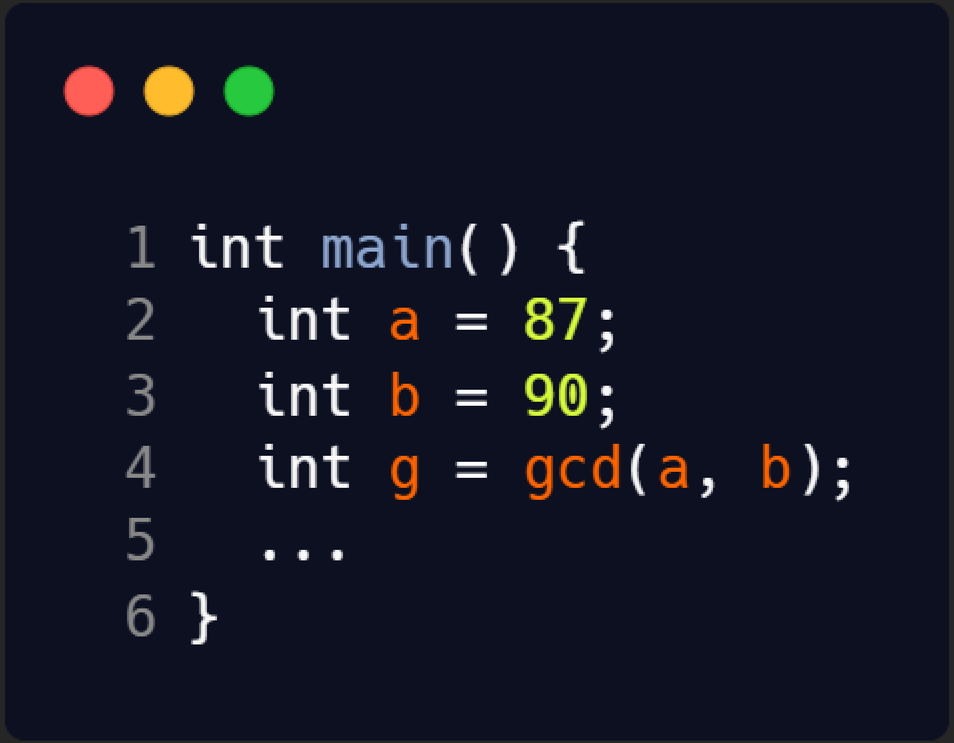


```
public main
main proc near
; __unwind {
endbr64
push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     dword ptr [rbp-0Ch], 87
mov     dword ptr [rbp-8], 90
mov     edx, [rbp-8]
mov     eax, [rbp-0Ch]
mov     esi, edx
mov     edi, eax
call    gcd
mov     [rbp-4], eax
mov     eax, [rbp-4]
mov     esi, eax
lea     rdi, format     ; "%d\n"
mov     eax, 0
call    _printf
mov     eax, 0
leave
retn
; } // starts at 119C
main endp
```

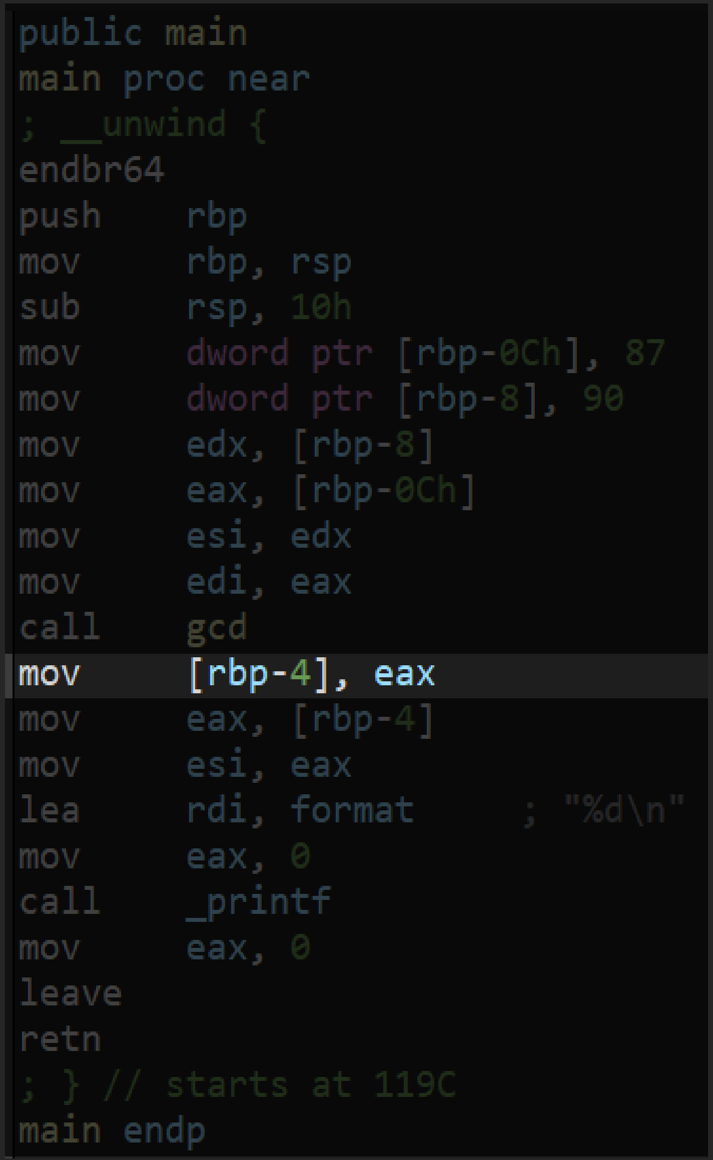
For security
無運算意義

Stack Frame
Prologue
(下個章節講)

C -> x86



```c
int main() {
    int a = 87;
    int b = 90;
    int g = gcd(a, b);
    ...
}
```

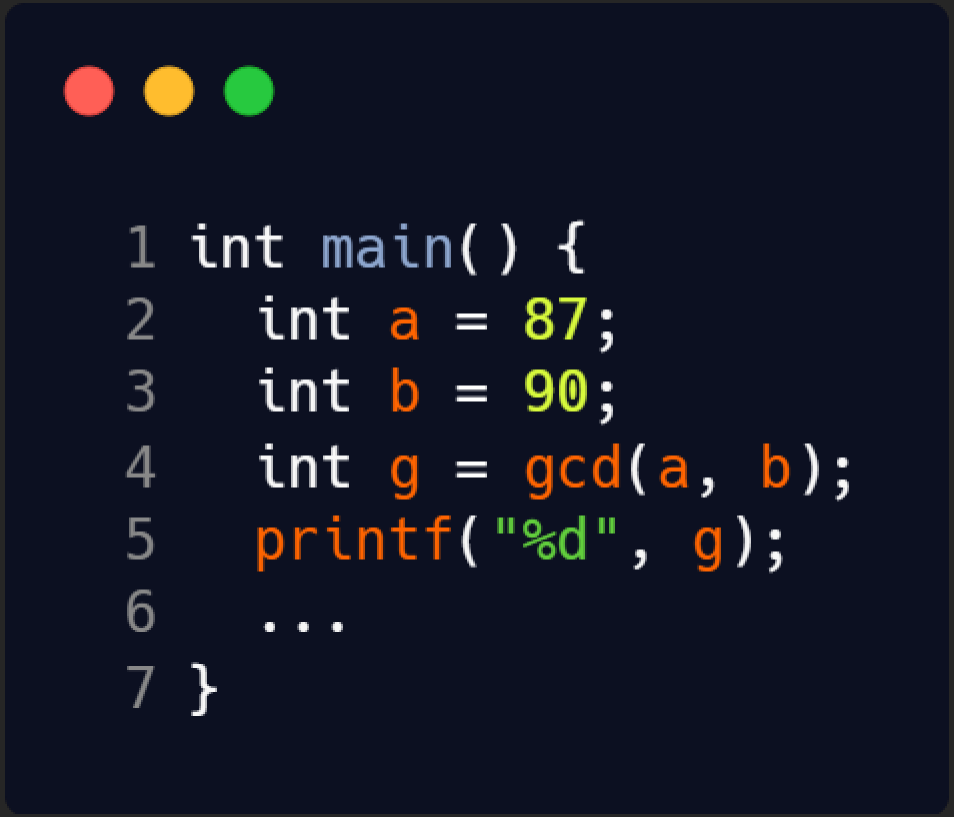


```
public main
main proc near
; __unwind {
endbr64
push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     dword ptr [rbp-0Ch], 87
mov     dword ptr [rbp-8], 90
mov     edx, [rbp-8]
mov     eax, [rbp-0Ch]
mov     esi, edx
mov     edi, eax
call    gcd
mov     [rbp-4], eax
mov     eax, [rbp-4]
mov     esi, eax
lea     rdi, format     ; "%d\n"
mov     eax, 0
call    _printf
mov     eax, 0
leave
retn
; } // starts at 119C
main endp
```

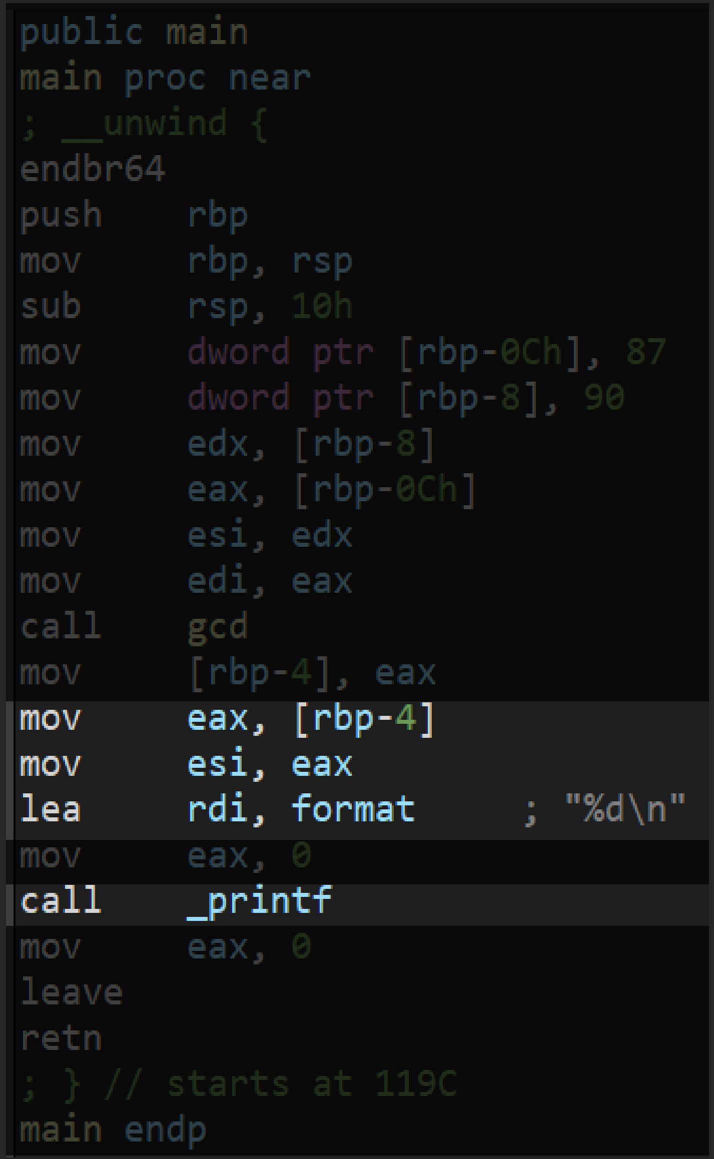
For security
無運算意義

Stack Frame
Prologue
(下個章節講)

C -> x86



```c
int main() {
  int a = 87;
  int b = 90;
  int g = gcd(a, b);
  printf("%d", g);
  ...
}
```

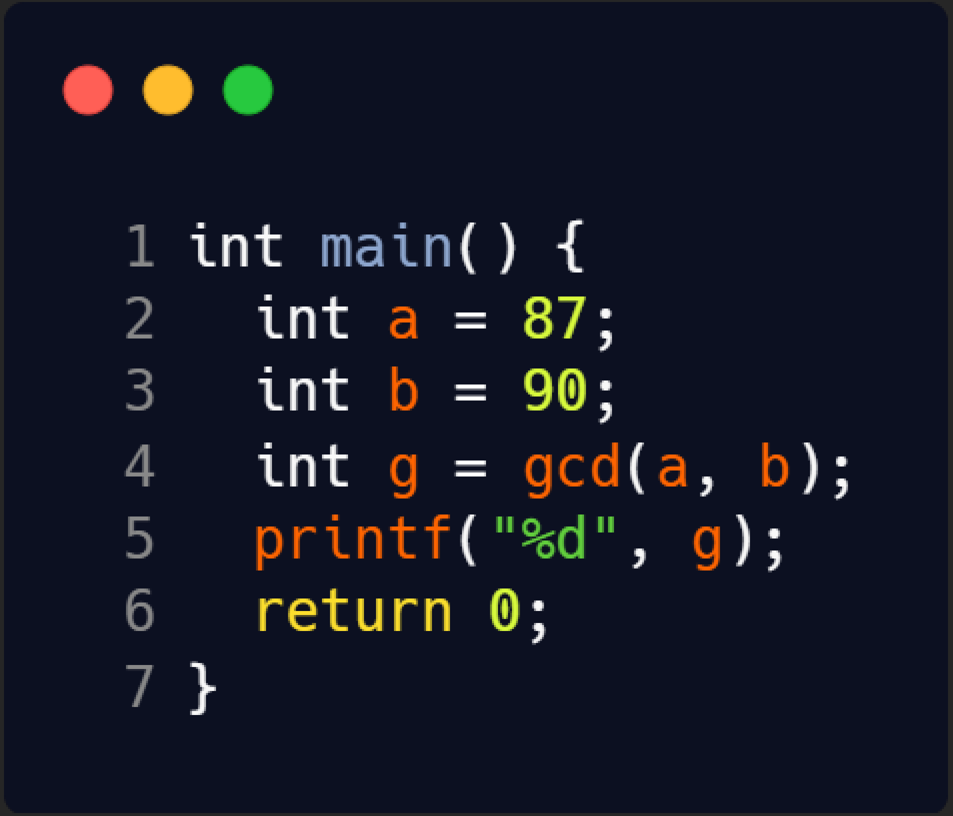


```
public main
main proc near
; __unwind {
endbr64
push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     dword ptr [rbp-0Ch], 87
mov     dword ptr [rbp-8], 90
mov     edx, [rbp-8]
mov     eax, [rbp-0Ch]
mov     esi, edx
mov     edi, eax
call    gcd
mov     [rbp-4], eax
mov     eax, [rbp-4]
mov     esi, eax
lea     rdi, format     ; "%d\n"
mov     eax, 0
call    _printf
mov     eax, 0
leave
retn
; } // starts at 119C
main endp
```

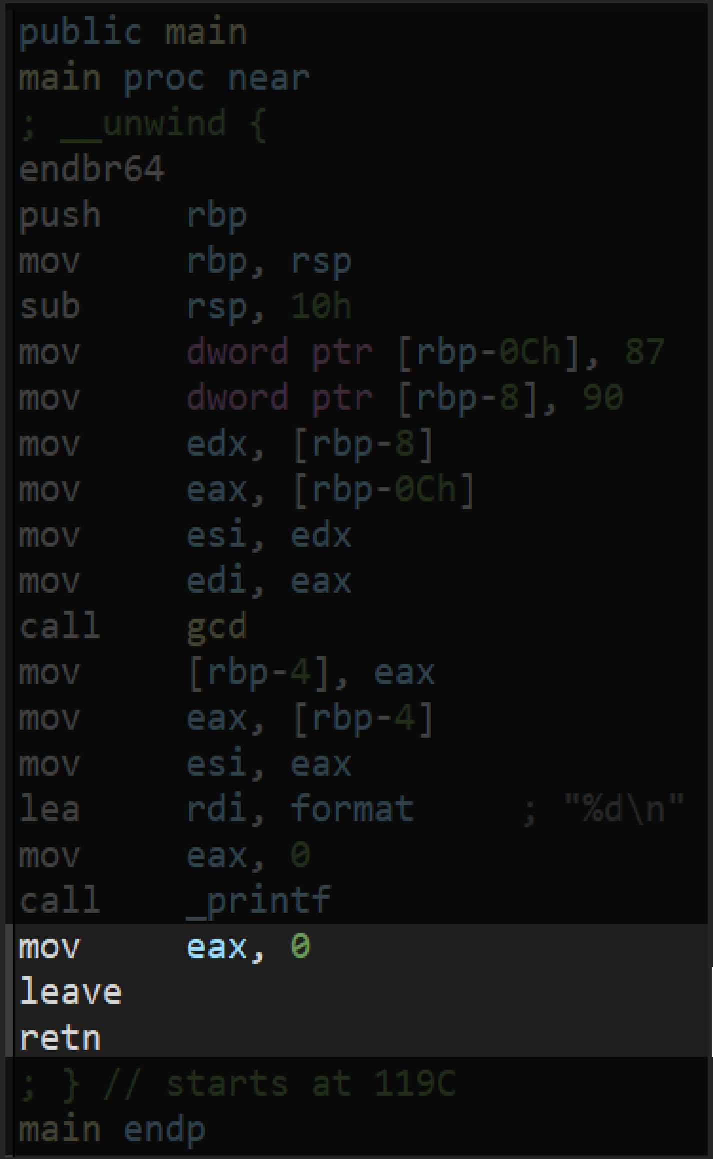
For security
無運算意義

Stack Frame
Prologue
(下個章節講)

C -> x86



```c
int main() {
   int a = 87;
   int b = 90;
   int g = gcd(a, b);
   printf("%d", g);
   return 0;
}
```



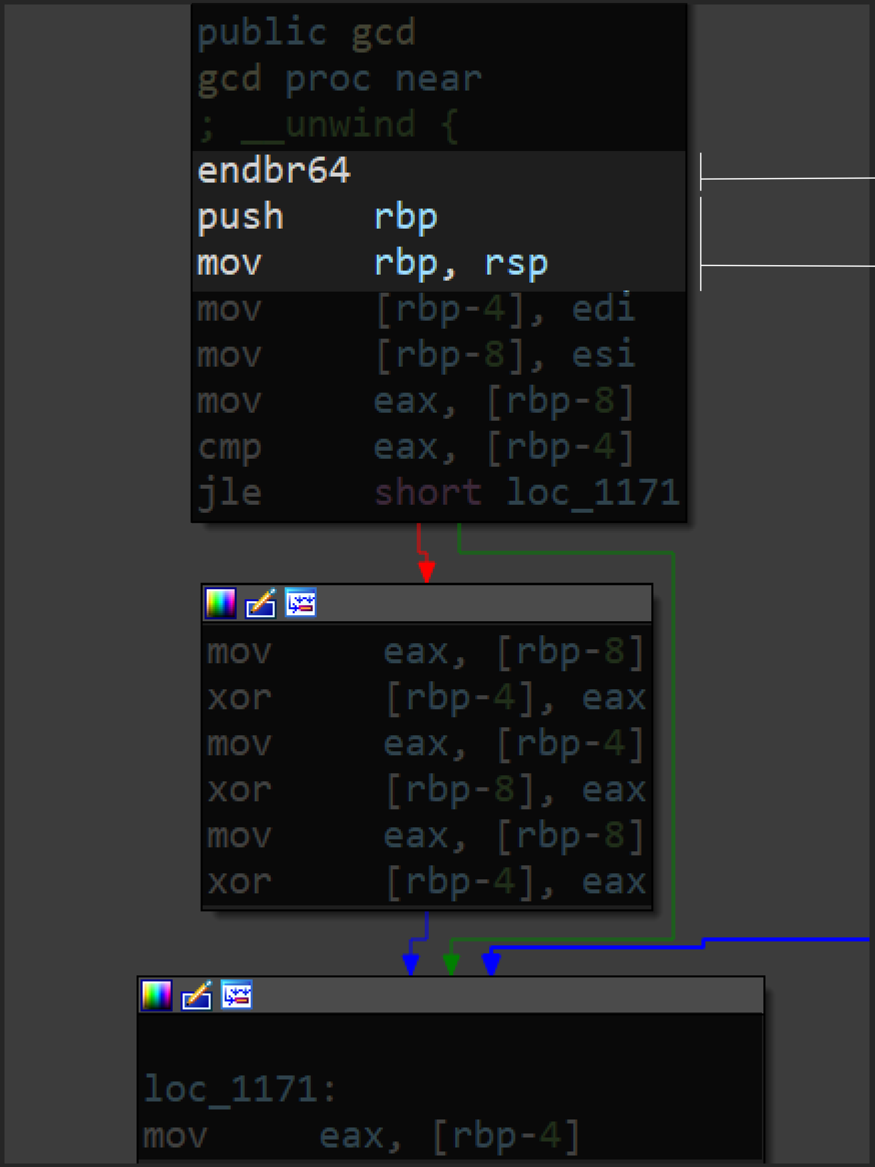
```asm
public main
main proc near
; __unwind {
endbr64
push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     dword ptr [rbp-0Ch], 87
mov     dword ptr [rbp-8], 90
mov     edx, [rbp-8]
mov     eax, [rbp-0Ch]
mov     esi, edx
mov     edi, eax
call    gcd
mov     [rbp-4], eax
mov     eax, [rbp-4]
mov     esi, eax
lea     rdi, format     ; "%d\n"
mov     eax, 0
call    _printf
mov     eax, 0
leave
retn
; } // starts at 119C
main endp
```

For security
無運算意義

Stack Frame
Prologue
(下個章節講)

Stack Frame
Epilogue
(下個章節講)

C -> x86



```
public gcd
gcd proc near
; __unwind {
endbr64
push    rbp
mov     rbp, rsp
mov     [rbp-4], edi
mov     [rbp-8], esi
mov     eax, [rbp-8]
cmp     eax, [rbp-4]
jle     short loc_1171
```

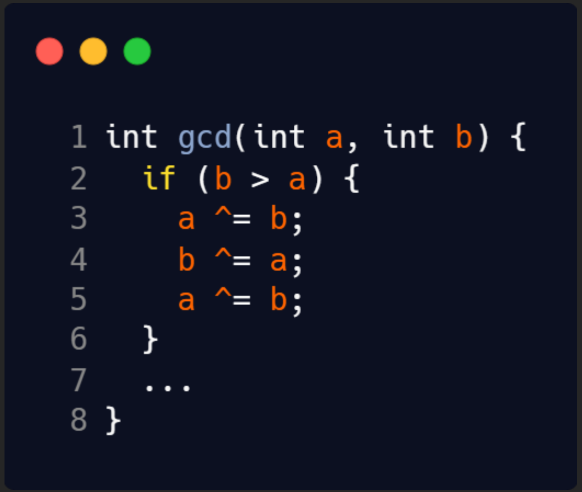
```
mov     eax, [rbp-8]
xor     [rbp-4], eax
mov     eax, [rbp-4]
xor     [rbp-8], eax
mov     eax, [rbp-8]
xor     [rbp-4], eax
```

```
loc_1171:
mov     eax, [rbp-4]
```

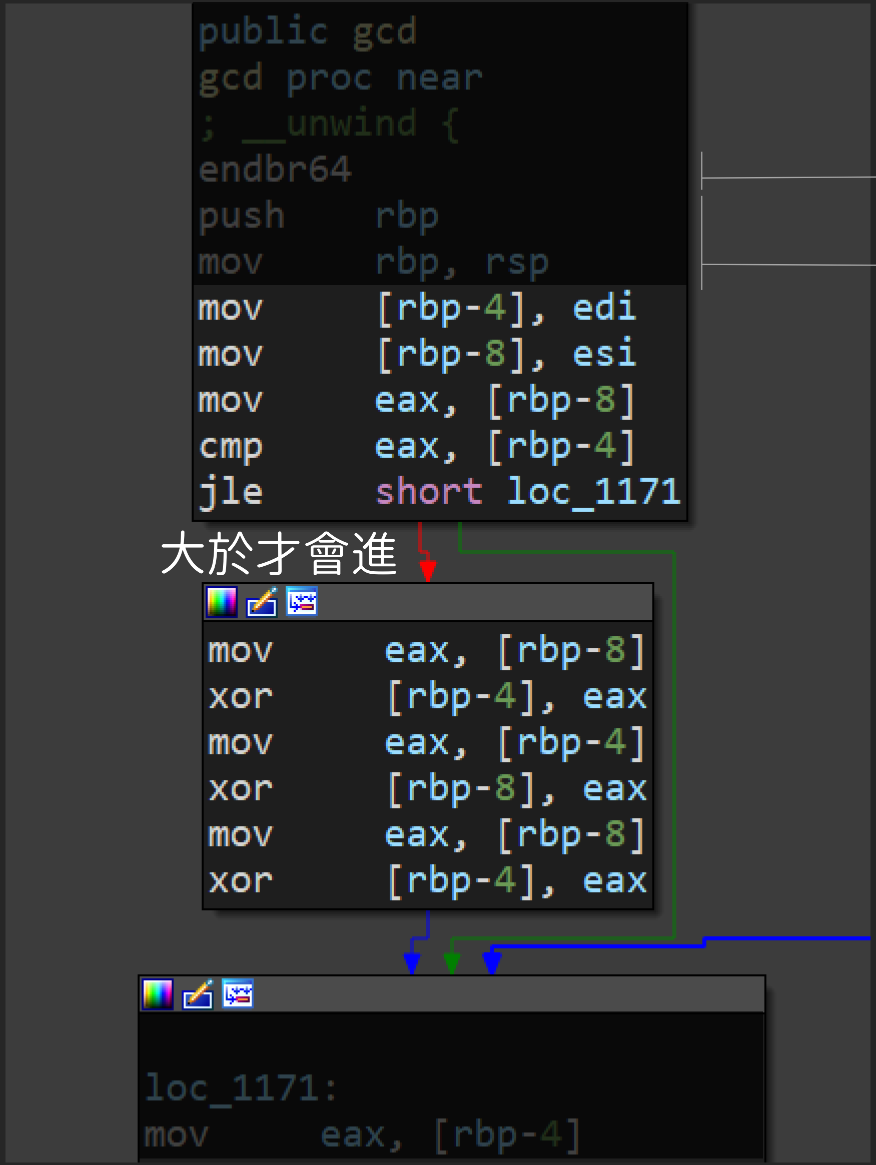
For security
無運算意義

Stack Frame
Prologue
(下個章節講)

C -> x86



```c
int gcd(int a, int b) {
  if (b > a) {
    a ^= b;
    b ^= a;
    a ^= b;
  }
  ...
}
```



```
public gcd
gcd proc near
; __unwind {
endbr64
push    rbp
mov     rbp, rsp
mov     [rbp-4], edi
mov     [rbp-8], esi
mov     eax, [rbp-8]
cmp     eax, [rbp-4]
jle     short loc_1171
```

```
mov     eax, [rbp-8]
xor     [rbp-4], eax
mov     eax, [rbp-4]
xor     [rbp-8], eax
mov     eax, [rbp-8]
xor     [rbp-4], eax
```

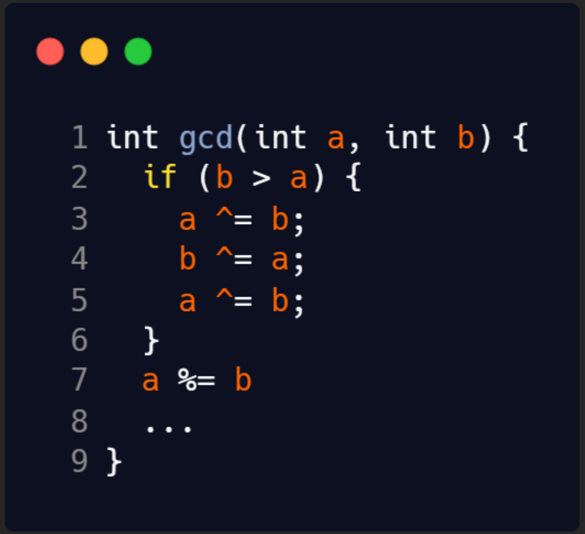
```
loc_1171:
mov     eax, [rbp-4]
```

大於才會進

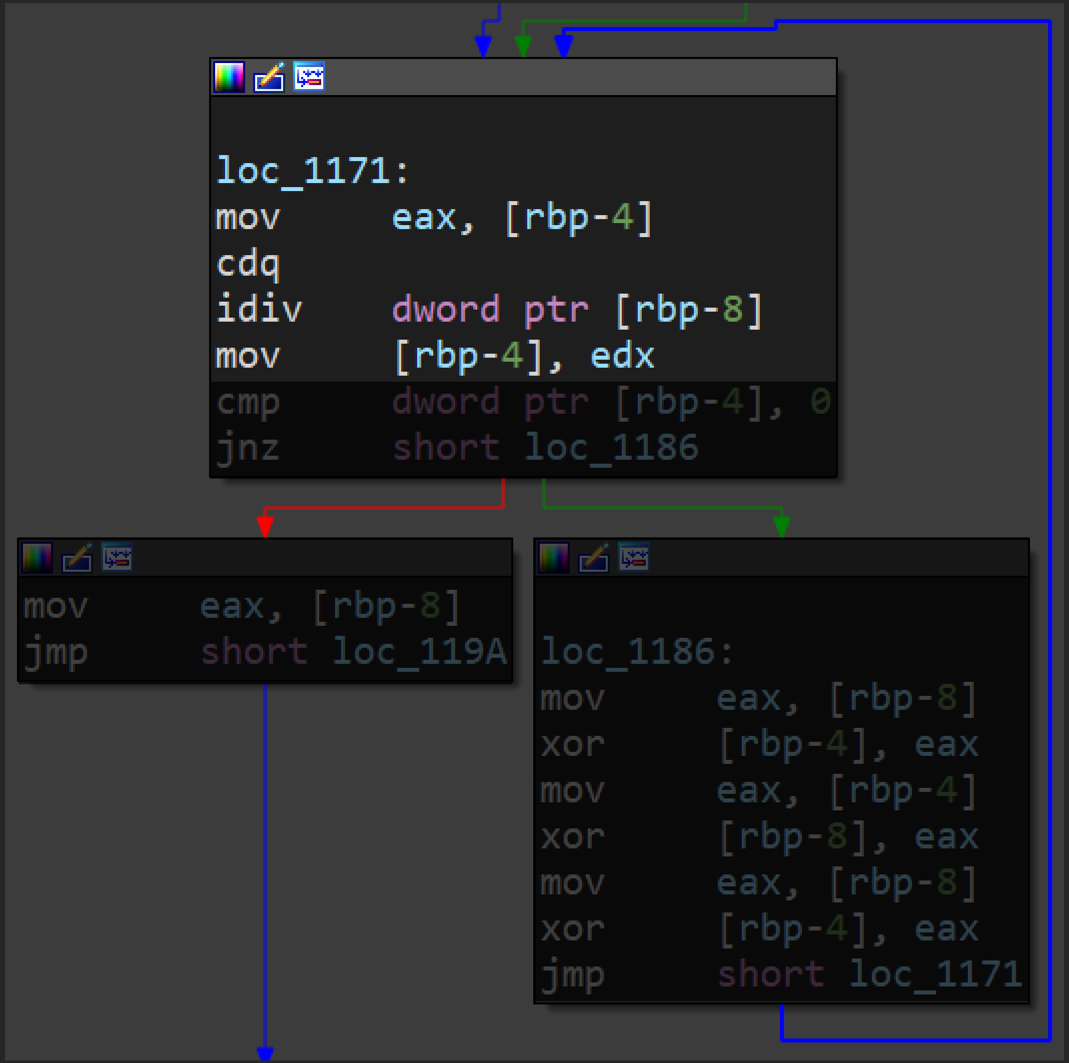
For security
無運算意義

Stack Frame
Prologue
(下個章節講)

C -> x86



```c
int gcd(int a, int b) {
  if (b > a) {
    a ^= b;
    b ^= a;
    a ^= b;
  }
  a %= b
  ...
}
```

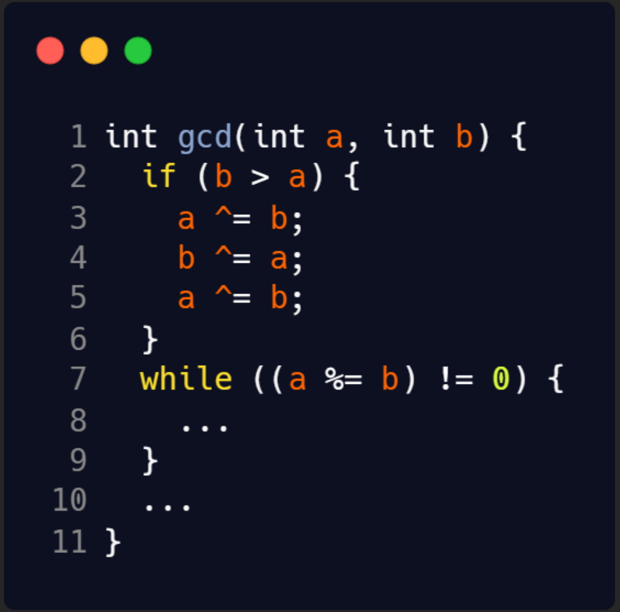


```
loc_1171:
mov     eax, [rbp-4]
cdq
idiv    dword ptr [rbp-8]
mov     [rbp-4], edx
cmp     dword ptr [rbp-4], 0
jnz     short loc_1186
```

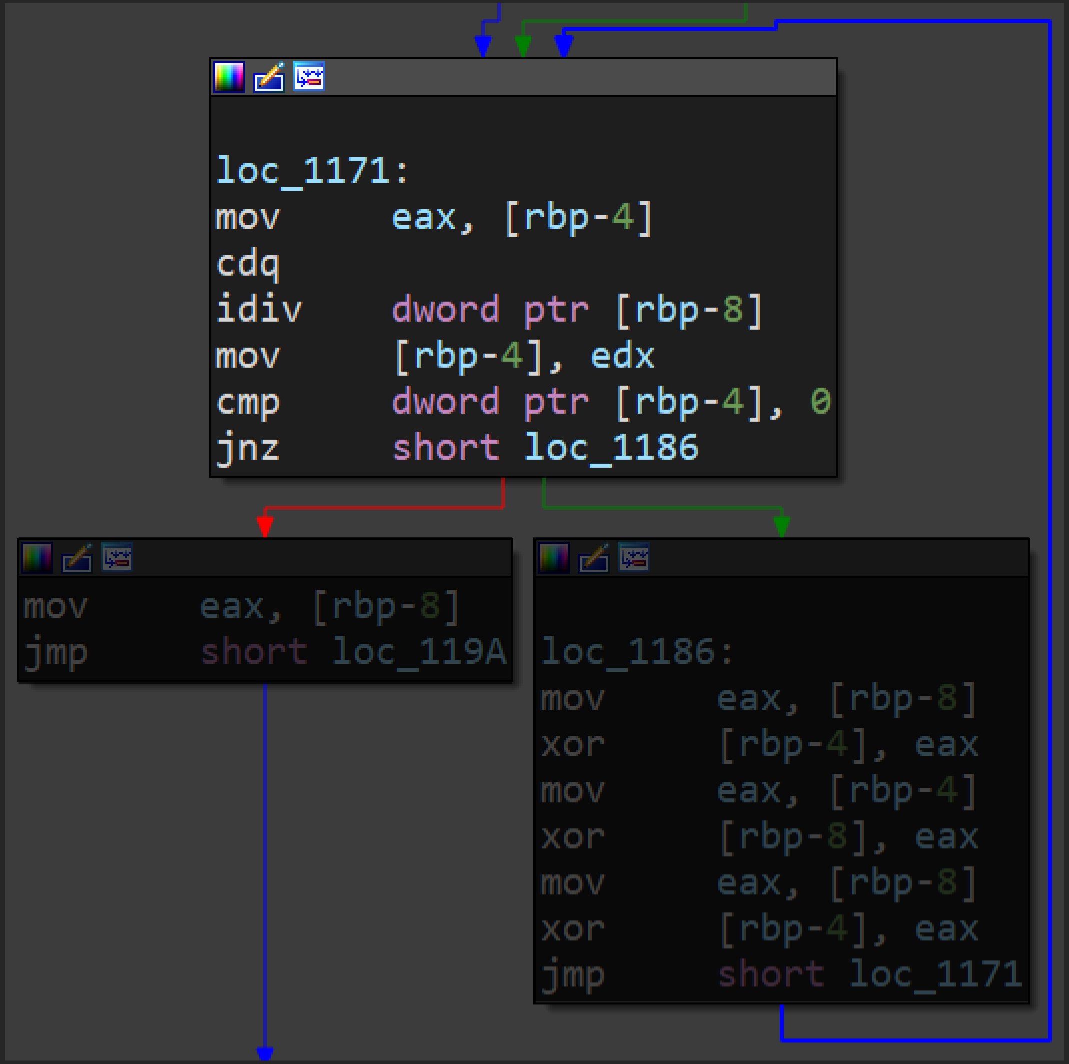
```
mov     eax, [rbp-8]
jmp     short loc_119A
```

```
loc_1186:
mov     eax, [rbp-8]
xor     [rbp-4], eax
mov     eax, [rbp-4]
xor     [rbp-8], eax
mov     eax, [rbp-8]
xor     [rbp-4], eax
jmp     short loc_1171
```

C -> x86



```c
int gcd(int a, int b) {
  if (b > a) {
    a ^= b;
    b ^= a;
    a ^= b;
  }
  while ((a %= b) != 0) {
    ...
  }
  ...
}
```

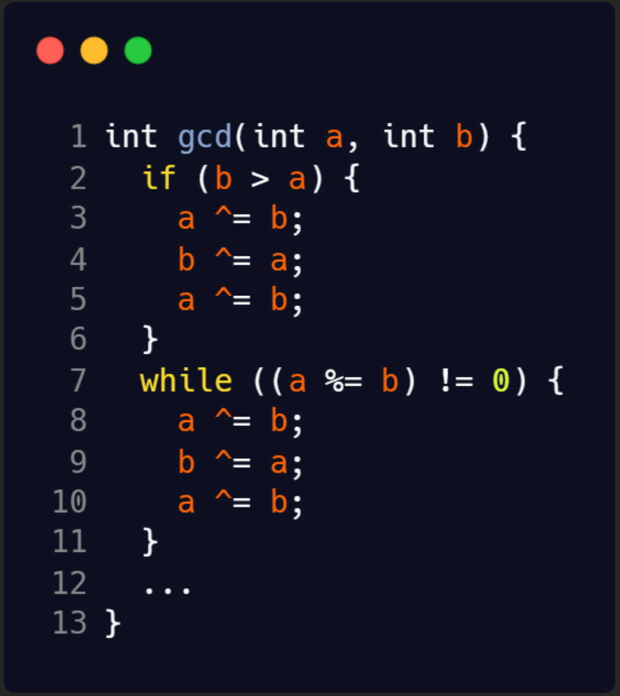


```
loc_1171:
mov     eax, [rbp-4]
cdq
idiv    dword ptr [rbp-8]
mov     [rbp-4], edx
cmp     dword ptr [rbp-4], 0
jnz     short loc_1186
```

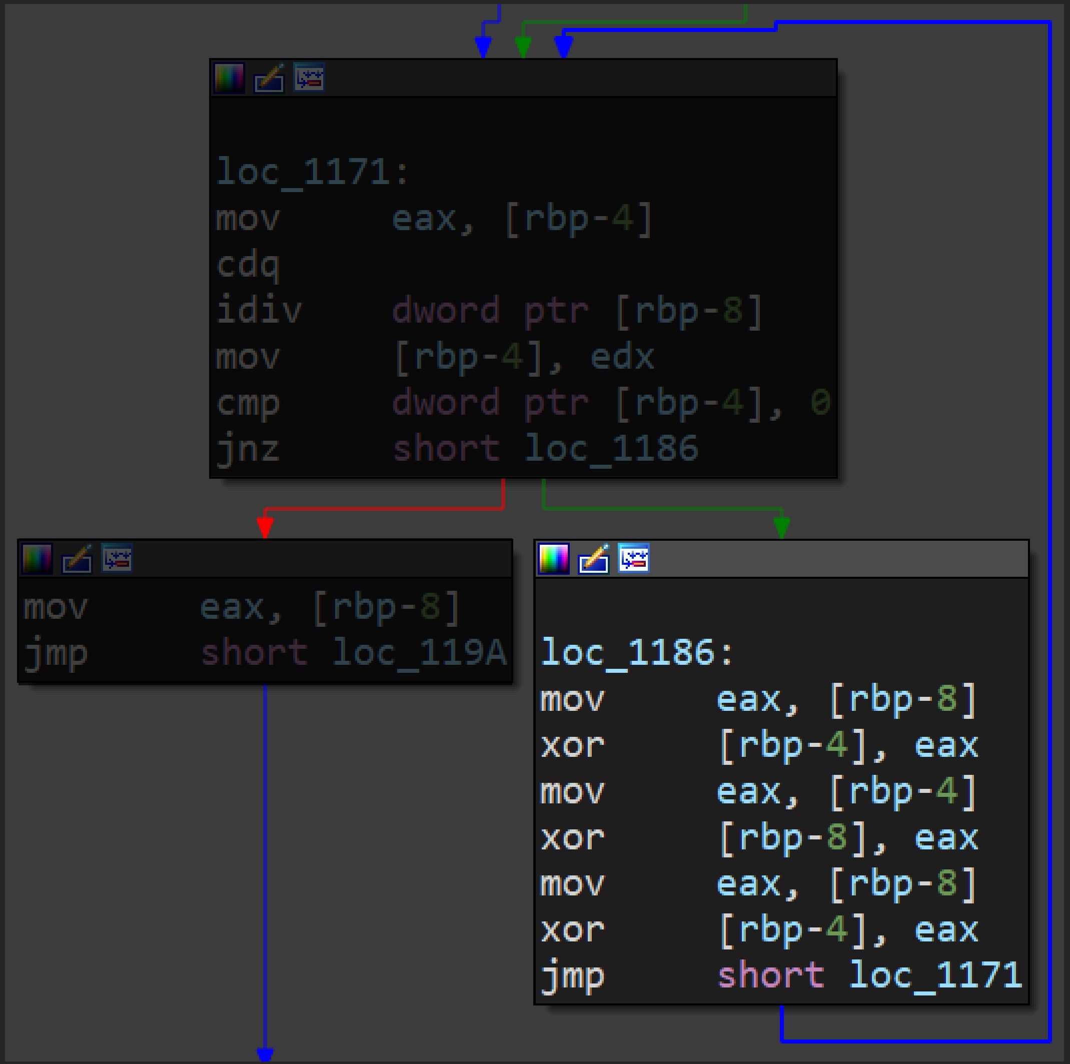
```
mov     eax, [rbp-8]
jmp     short loc_119A
```

```
loc_1186:
mov     eax, [rbp-8]
xor     [rbp-4], eax
mov     eax, [rbp-4]
xor     [rbp-8], eax
mov     eax, [rbp-8]
xor     [rbp-4], eax
jmp     short loc_1171
```

C -> x86



```c
int gcd(int a, int b) {
  if (b > a) {
    a ^= b;
    b ^= a;
    a ^= b;
  }
  while ((a %= b) != 0) {
    a ^= b;
    b ^= a;
    a ^= b;
  }
  ...
}
```

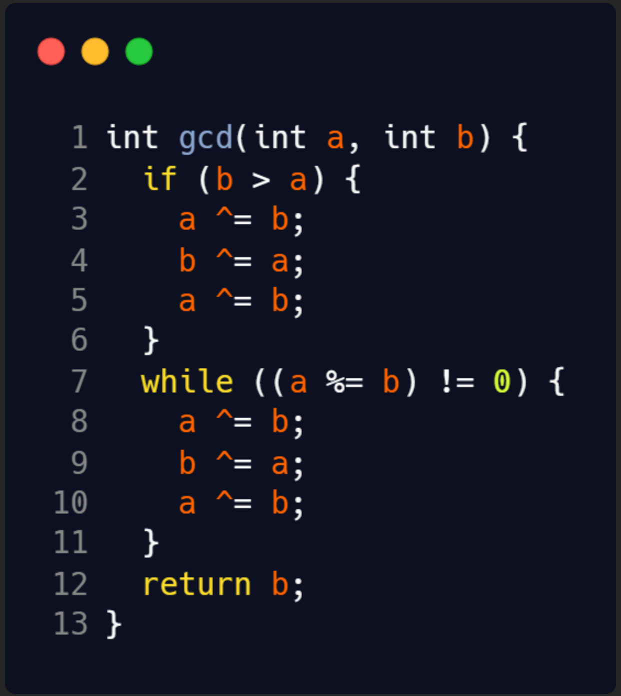


```
loc_1171:
mov     eax, [rbp-4]
cdq
idiv    dword ptr [rbp-8]
mov     [rbp-4], edx
cmp     dword ptr [rbp-4], 0
jnz     short loc_1186
```

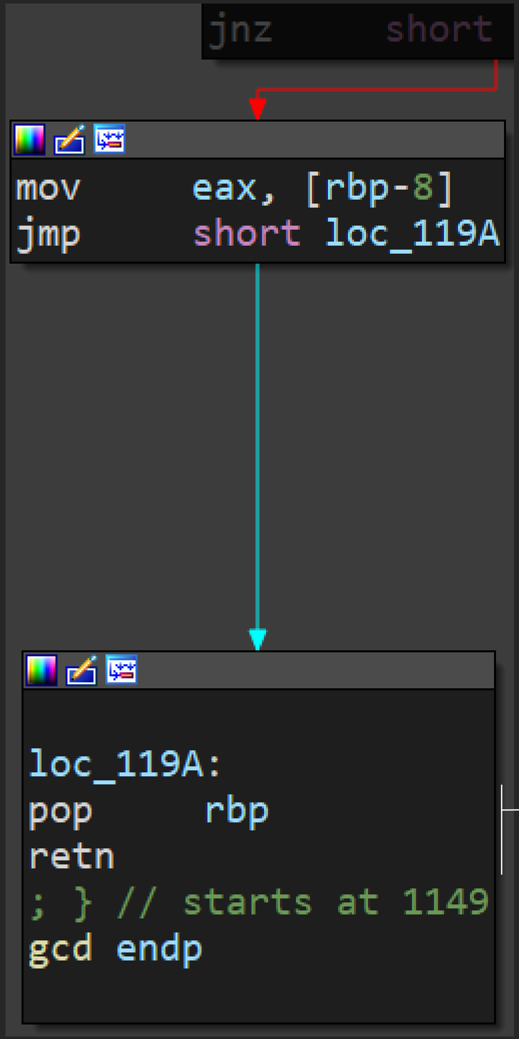
```
mov     eax, [rbp-8]
jmp     short loc_119A
```

```
loc_1186:
mov     eax, [rbp-8]
xor     [rbp-4], eax
mov     eax, [rbp-4]
xor     [rbp-8], eax
mov     eax, [rbp-8]
xor     [rbp-4], eax
jmp     short loc_1171
```

C -> x86



```c
int gcd(int a, int b) {
  if (b > a) {
    a ^= b;
    b ^= a;
    a ^= b;
  }
  while ((a %= b) != 0) {
    a ^= b;
    b ^= a;
    a ^= b;
  }
  return b;
}
```



```
jnz     short

mov     eax, [rbp-8]
jmp     short loc_119A

loc_119A:
pop     rbp
retn
; } // starts at 1149
gcd endp
```

Stack Frame Epilogue (下個章節講)

C -> x86

- 小總結
- 目前為止, 除了和 Stack Frame 相關的組語, 應該都能看懂了
- if 會往前跳; 迴圈會往回跳

Stack Frame

Stack Frame

- Q1: 函數都是以 RSP 或 RBP 來定位區域變數，那怎麼區別不同函數的區域變數?

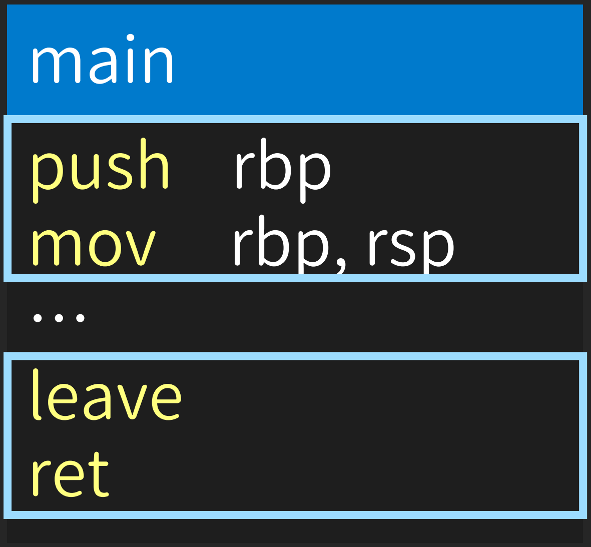
- Q2: 呼叫函數後, RIP 就從 A 函數跑到 B 函數了，要怎麼 return 回 A 函數?

- 如果不知道答案，那你就需要看一下這章

Stack Frame

- 不同區域會有不同的 Stack Frame
  - 裡面存放著區域變數

- 在 Function 的頭部和尾部, 有一些用來處理 Stack Frame 的指令
  - 頭部: Prologue
  - 尾部: Epilogue



```
main
push    rbp
mov     rbp, rsp
…
leave
ret
```

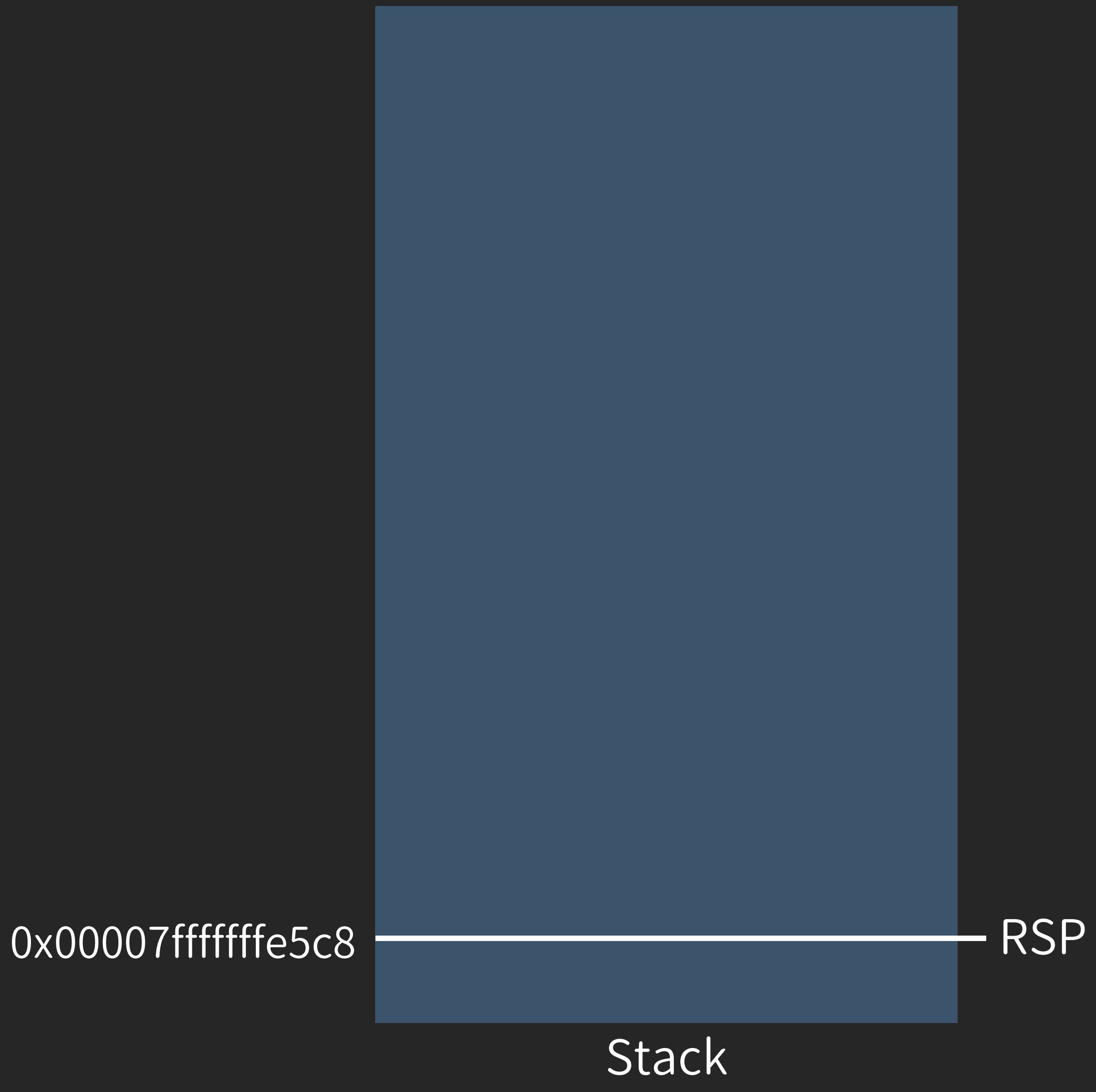
Stack Frame

main

```
push   rbp
mov    rbp, rsp
sub    rsp, 20h
...
call   function1
leave
ret
```

function1

```
push   rbp
mov    rbp, rsp
sub    rsp, 30h
...
leave
ret
```



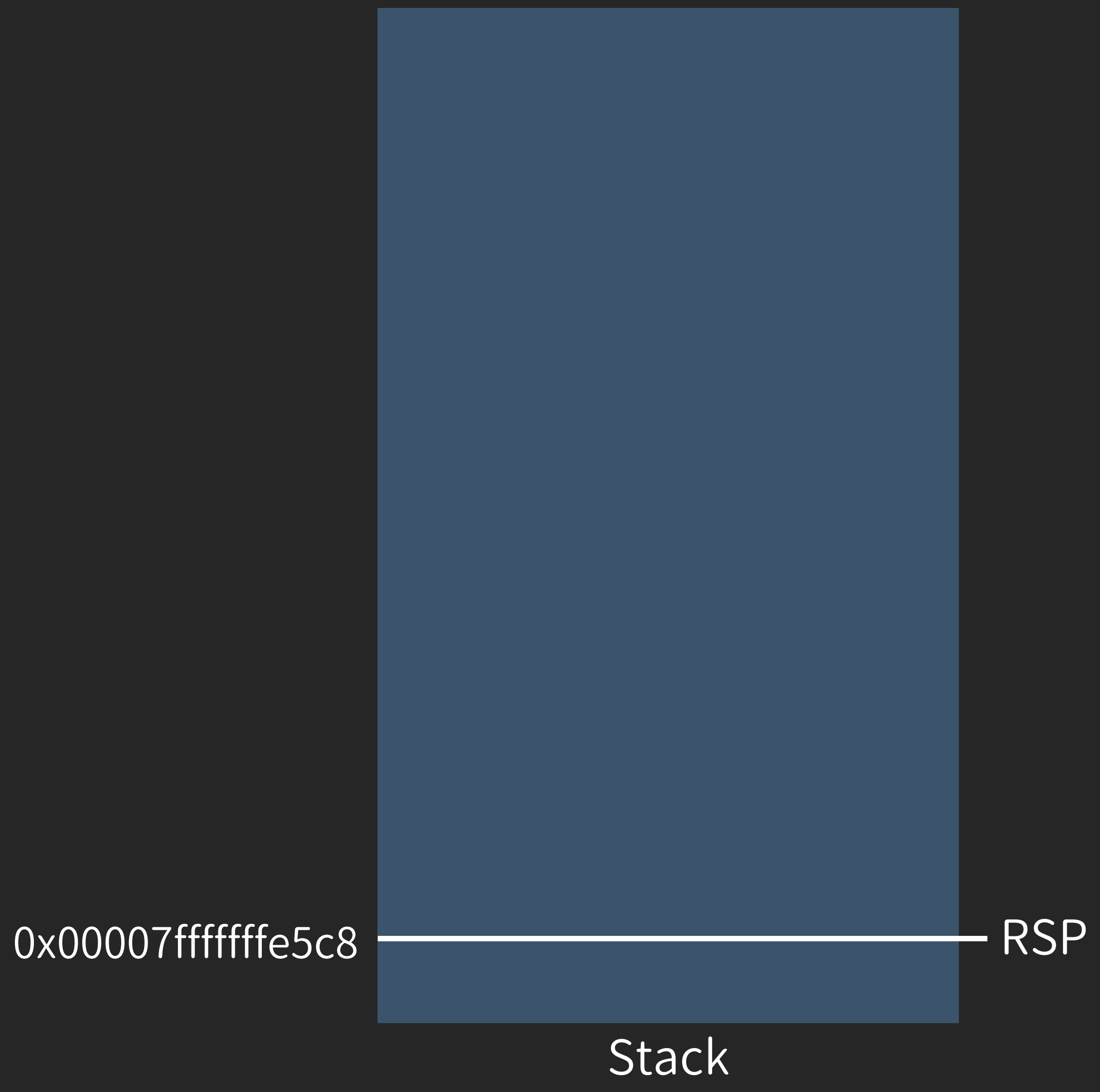
Stack Frame

main

```
push   rbp
mov    rbp, rsp
sub    rsp, 20h
…
call   function1
leave
ret
```

function1

```
push   rbp
mov    rbp, rsp
sub    rsp, 30h
…
leave
ret
```



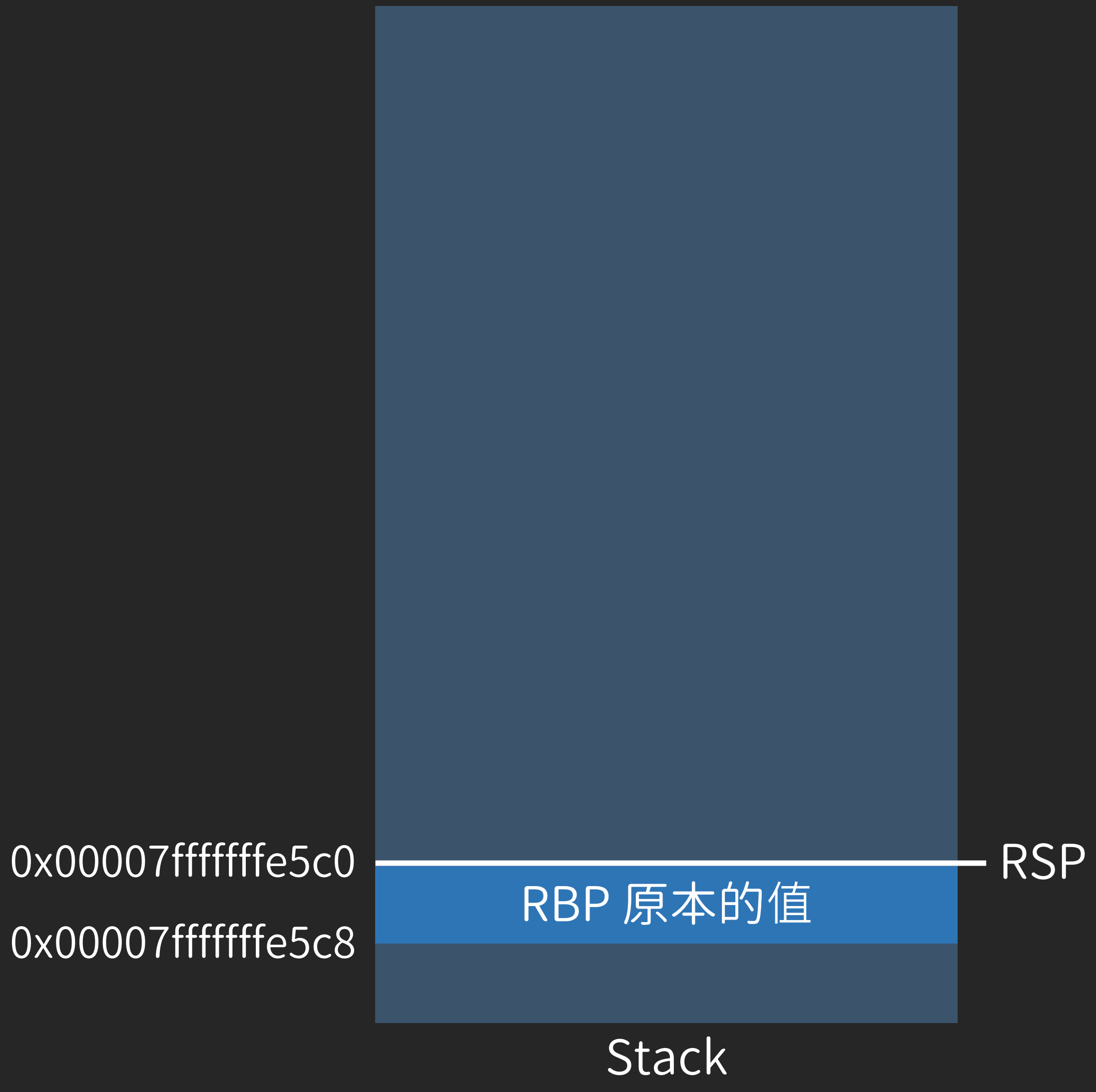
Stack Frame

main

```
push  rbp
mov   rbp, rsp
sub   rsp, 20h
…
call  function1
leave
ret
```

function1

```
push  rbp
mov   rbp, rsp
sub   rsp, 30h
…
leave
ret
```



0x00007fffffffe5c0

0x00007fffffffe5c8

RSP

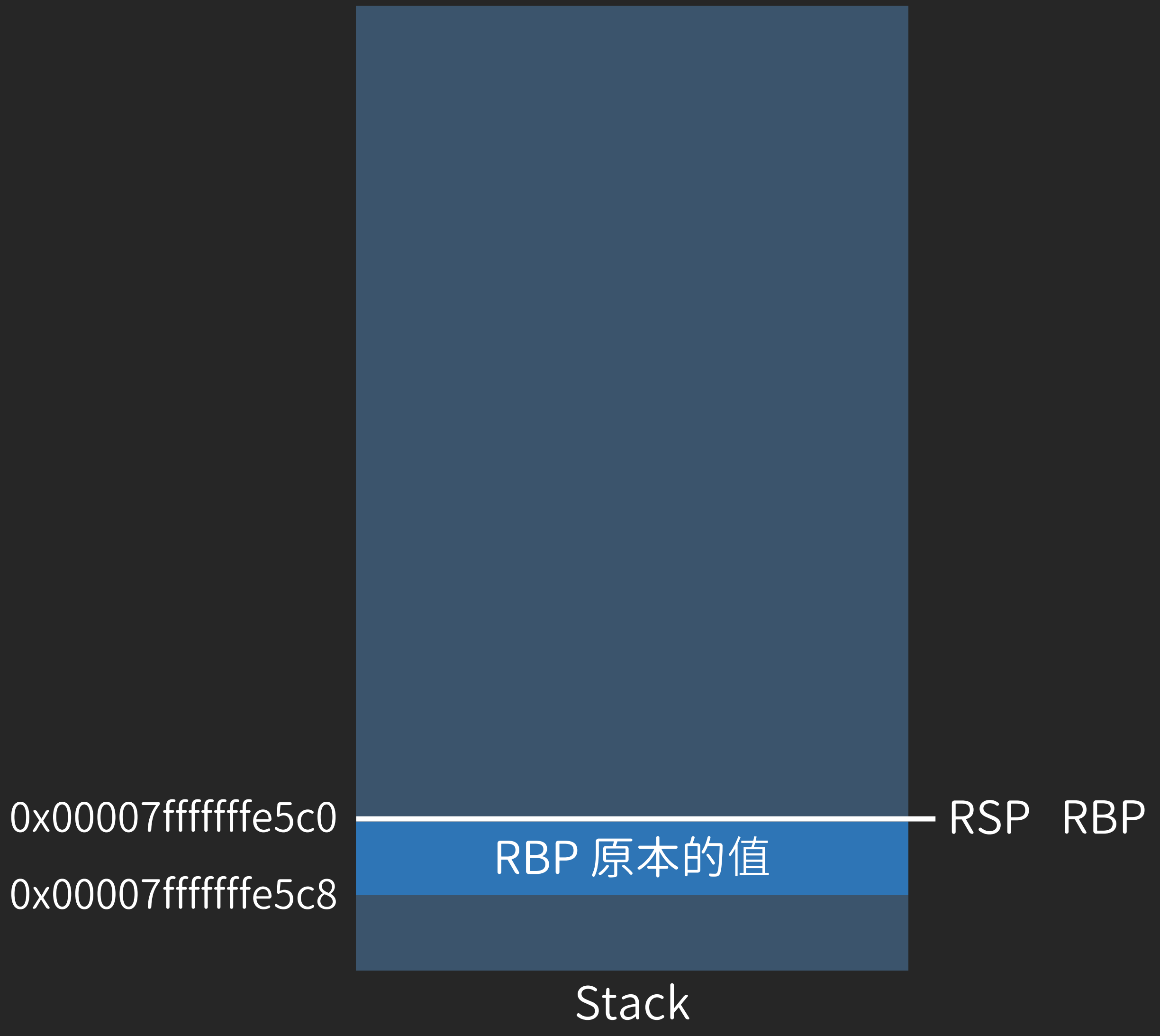
RBP 原本的值

Stack

Stack Frame

```
main
push  rbp
mov   rbp, rsp
sub   rsp, 20h
…
call  function1
leave
ret
```

```
function1
push  rbp
mov   rbp, rsp
sub   rsp, 30h
…
leave
ret
```



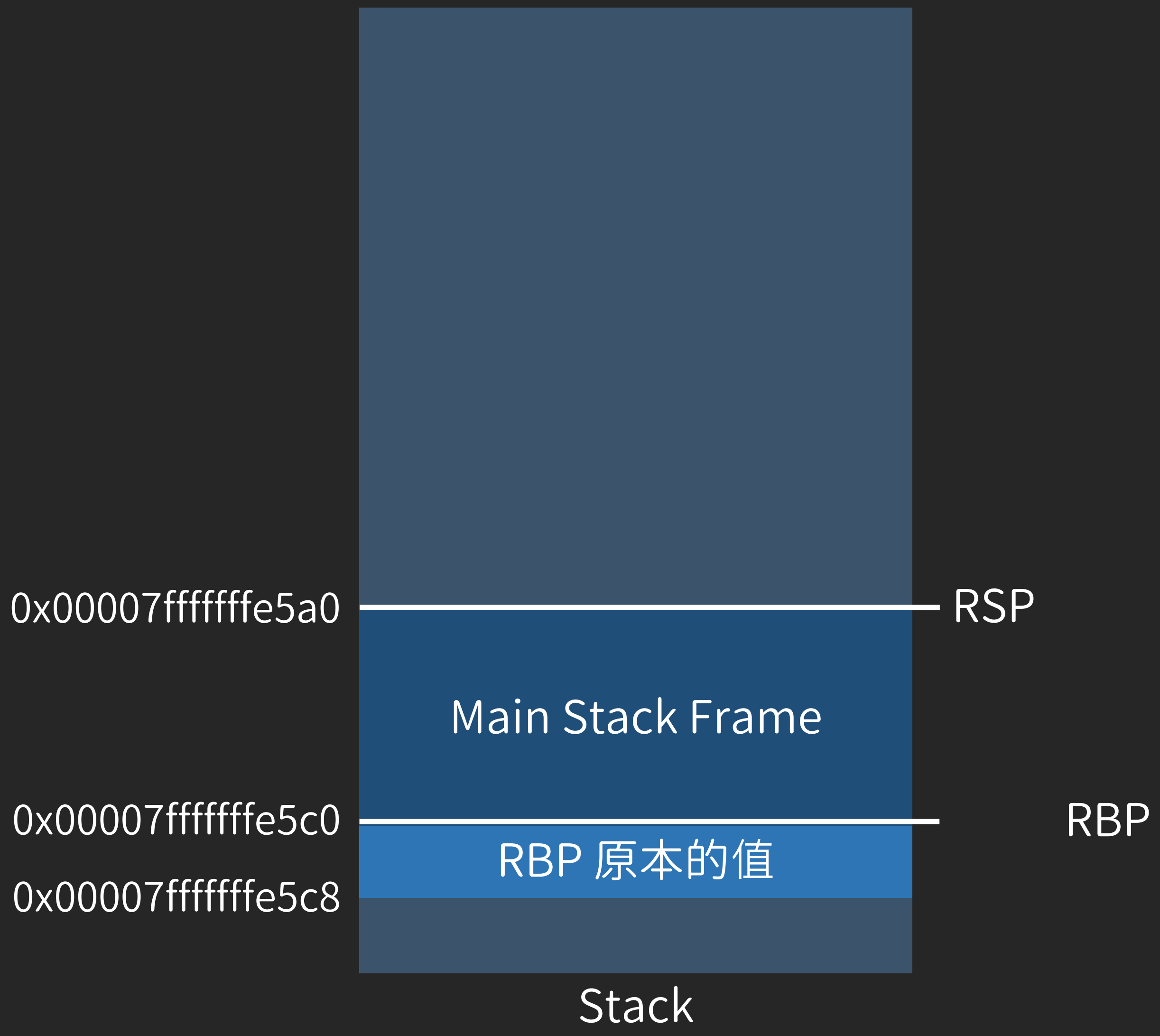
Stack Frame

main

```
push  rbp
mov   rbp, rsp
sub   rsp, 20h
…
call  function1
leave
ret
```

function1

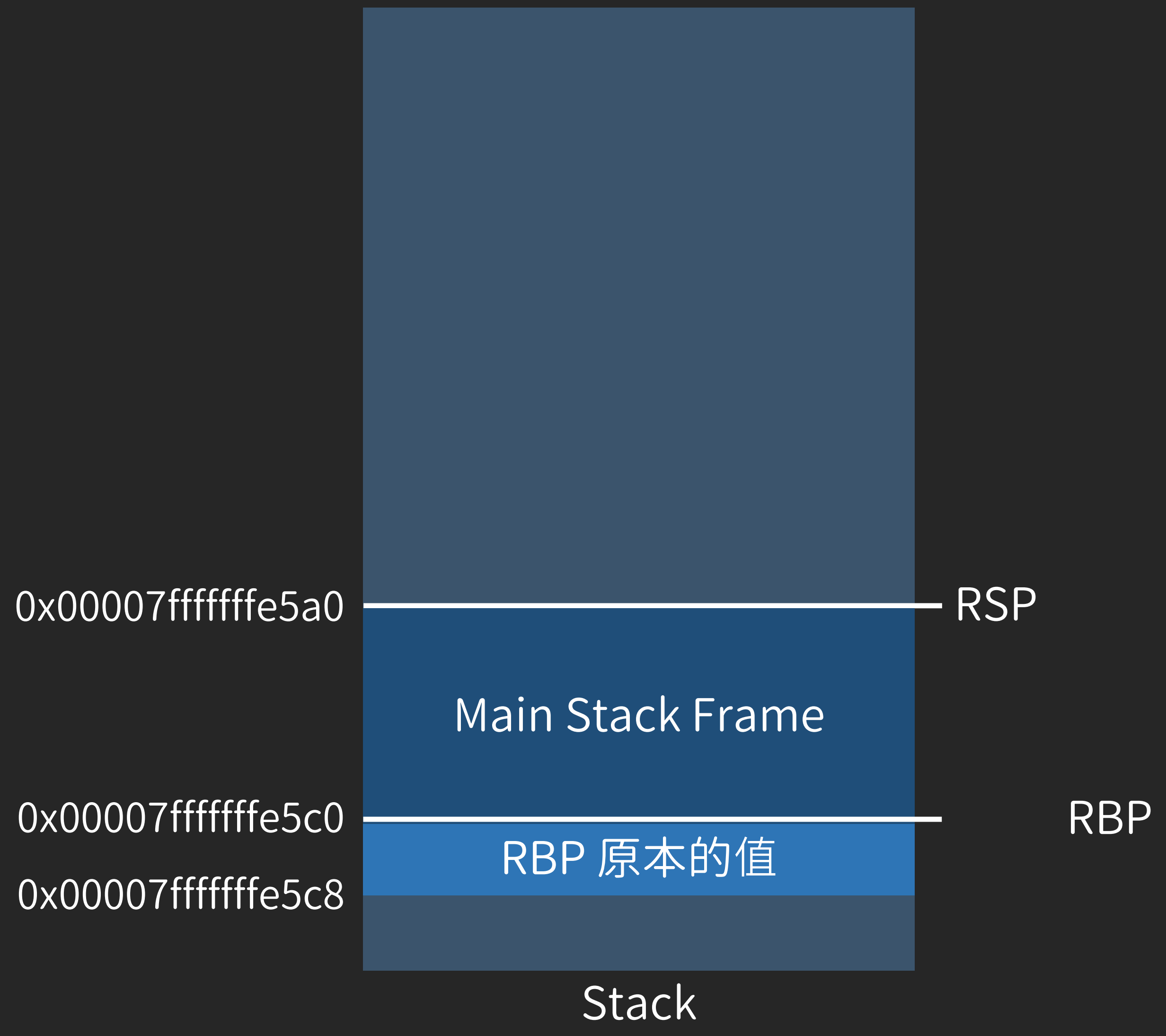
```
push  rbp
mov   rbp, rsp
sub   rsp, 30h
…
leave
ret
```



Stack Frame

```
main
         push   rbp
         mov    rbp, rsp
         sub    rsp, 20h
         …
         call   function1
0x401234 leave
         ret
```

```
function1
push   rbp
mov    rbp, rsp
sub    rsp, 30h
…
leave
ret
```



0x00007fffffffe5a0 — RSP

Main Stack Frame

0x00007fffffffe5c0 — RBP

RBP 原本的值

0x00007fffffffe5c8

Stack

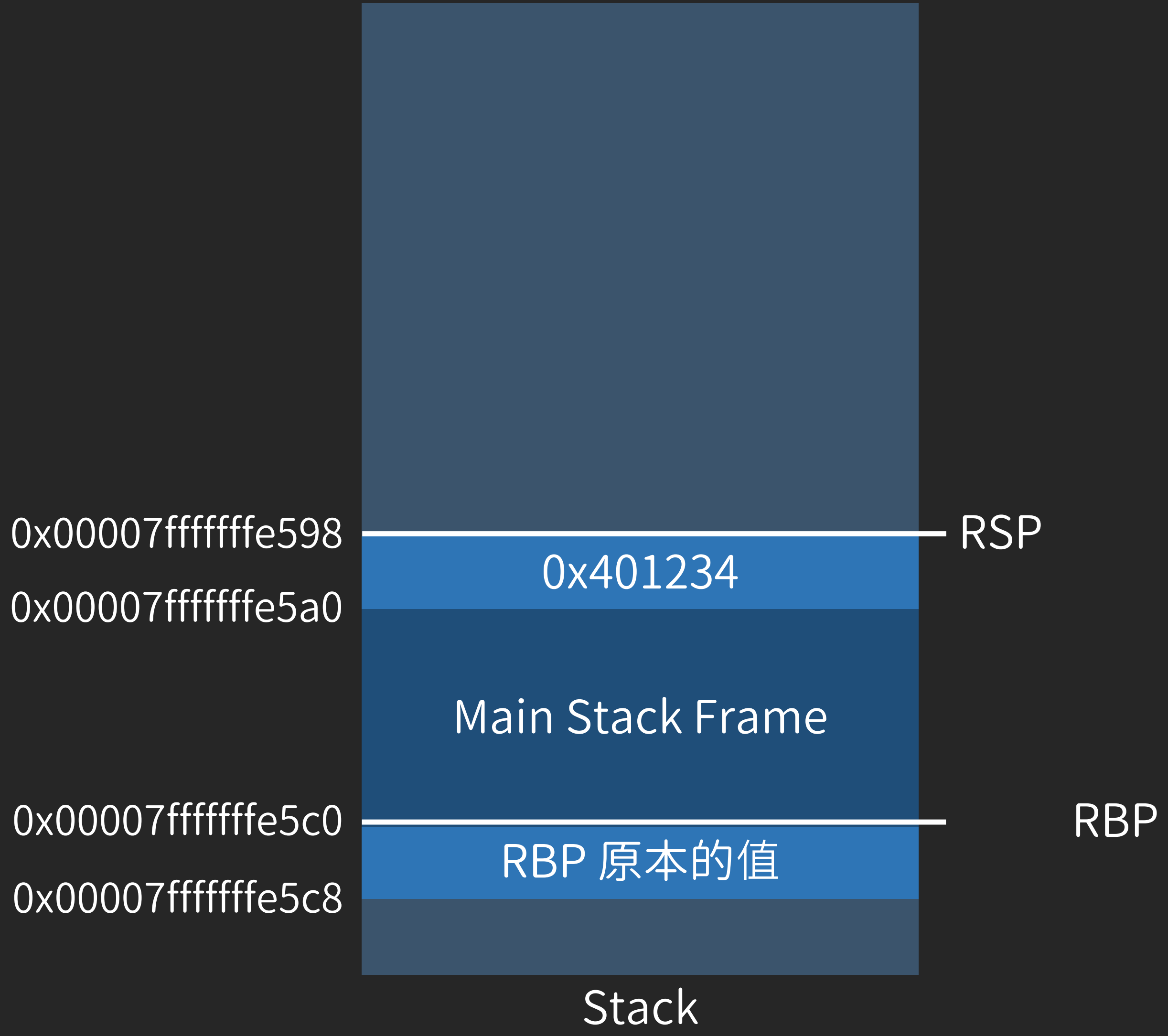
Stack Frame

main

```
         push  rbp
         mov   rbp, rsp
         sub   rsp, 20h
         …
         call  function1
0x401234 leave
         ret
```

function1

```
push  rbp
mov   rbp, rsp
sub   rsp, 30h
…
leave
ret
```



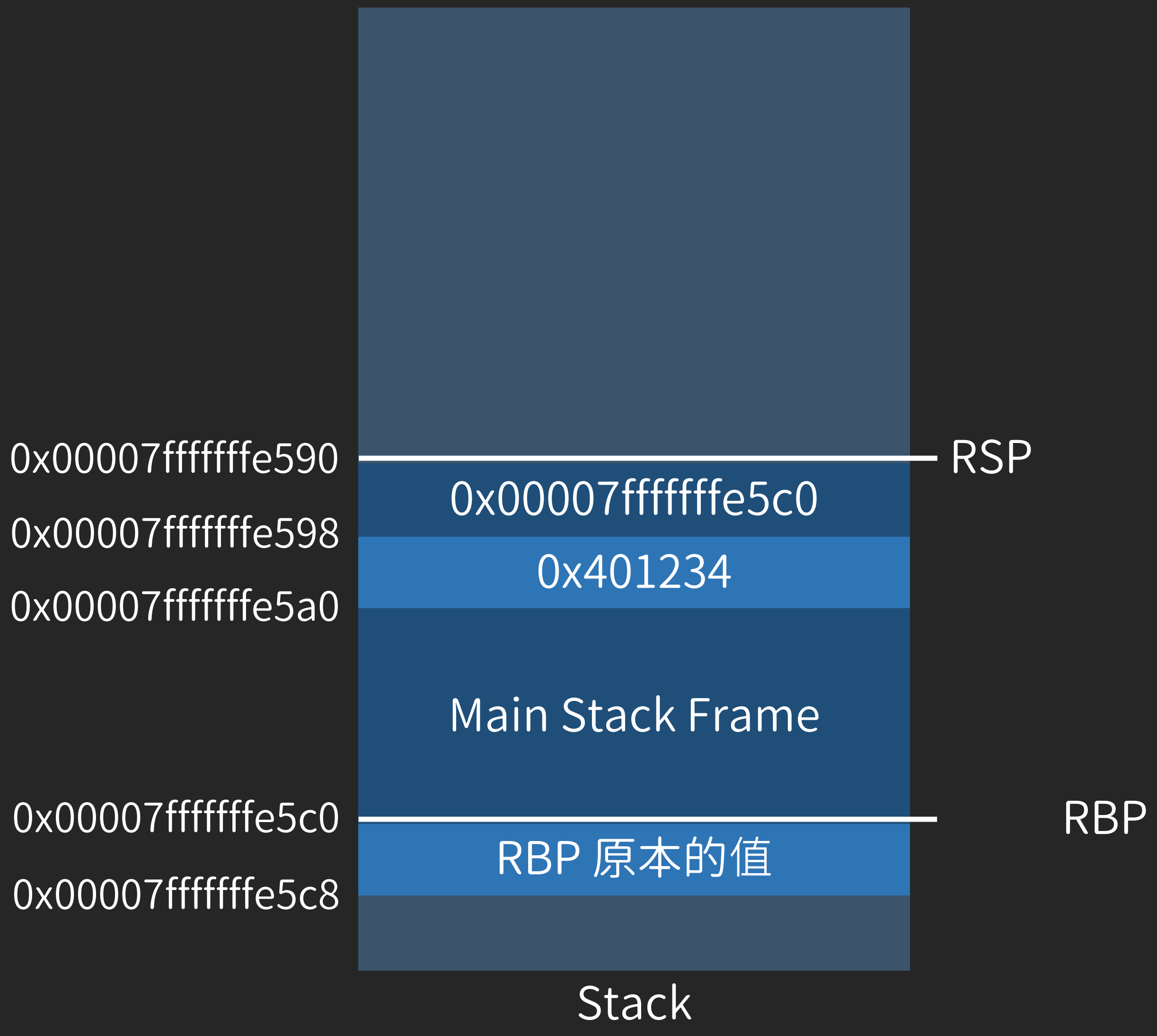
Stack Frame

main

```
         push  rbp
         mov   rbp, rsp
         sub   rsp, 20h
         …
         call  function1
0x401234 leave
         ret
```

function1

```
push  rbp
mov   rbp, rsp
sub   rsp, 30h
…
leave
ret
```



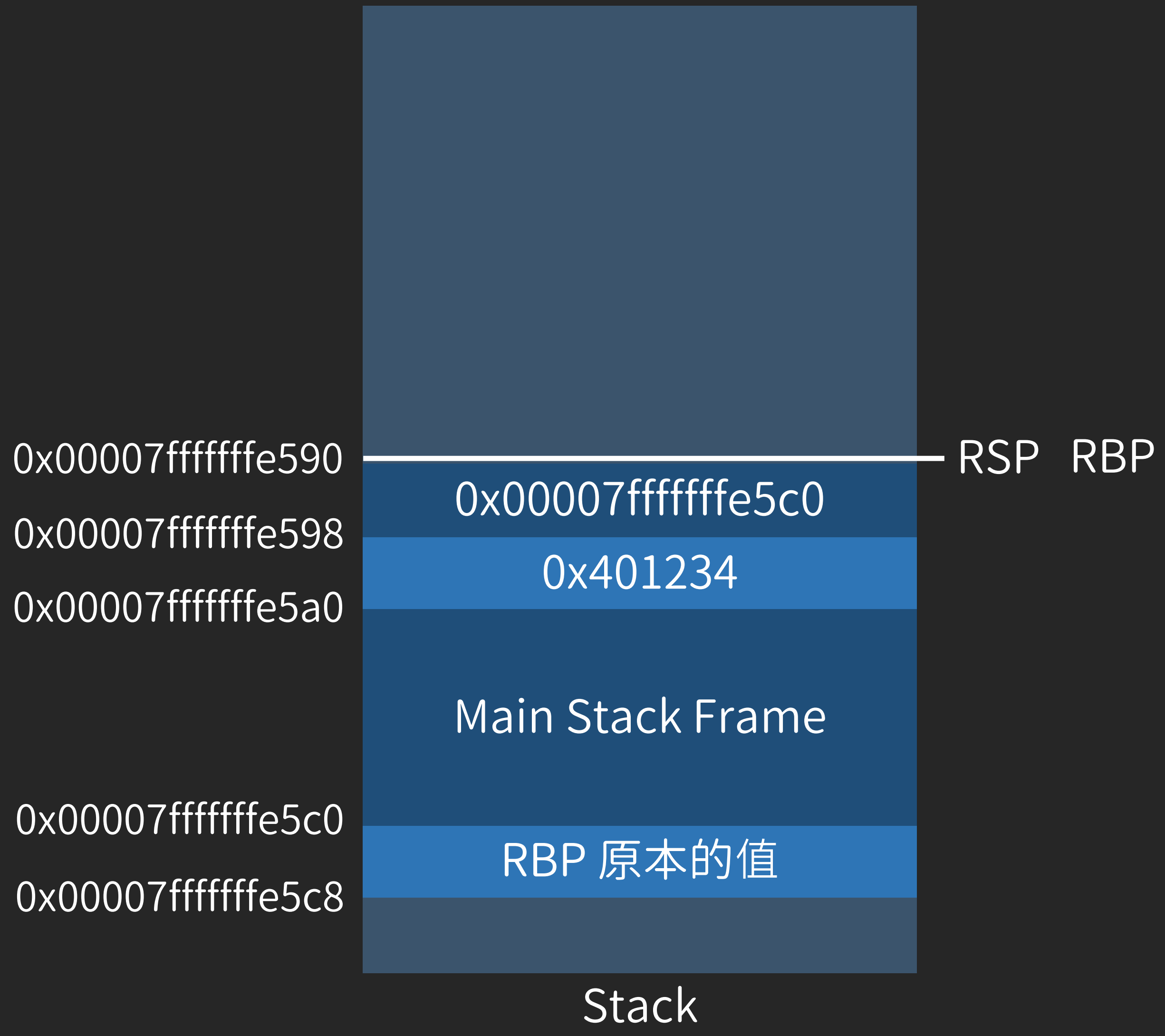
Stack Frame

main

```
         push  rbp
         mov   rbp, rsp
         sub   rsp, 20h
         …
         call  function1
0x401234 leave
         ret
```

function1

```
push  rbp
mov   rbp, rsp
sub   rsp, 30h
…
leave
ret
```



0x00007fffffffe590 — RSP RBP

0x00007fffffffe590: 0x00007fffffffe5c0

0x00007fffffffe598: 0x401234

0x00007fffffffe5a0

Main Stack Frame

0x00007fffffffe5c0: RBP 原本的值

0x00007fffffffe5c8

Stack

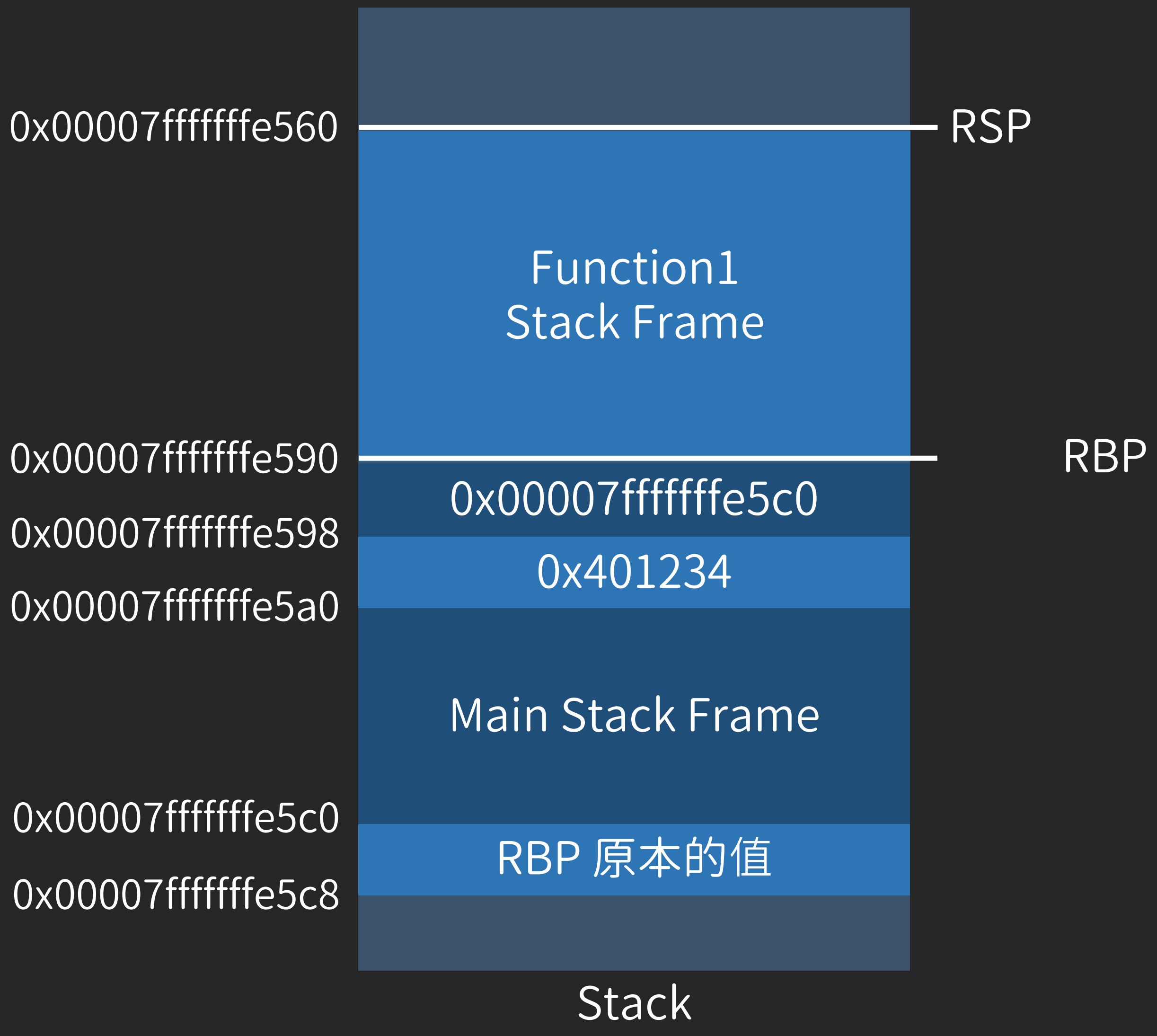
Stack Frame

main

```
         push   rbp
         mov    rbp, rsp
         sub    rsp, 20h
         …
         call   function1
0x401234 leave
         ret
```

function1

```
push   rbp
mov    rbp, rsp
sub    rsp, 30h
…
leave
ret
```



0x00007fffffffe560

RSP

Function1
Stack Frame

0x00007fffffffe590

RBP

0x00007fffffffe5c0

0x00007fffffffe598

0x401234

0x00007fffffffe5a0

Main Stack Frame

0x00007fffffffe5c0

RBP 原本的值

0x00007fffffffe5c8

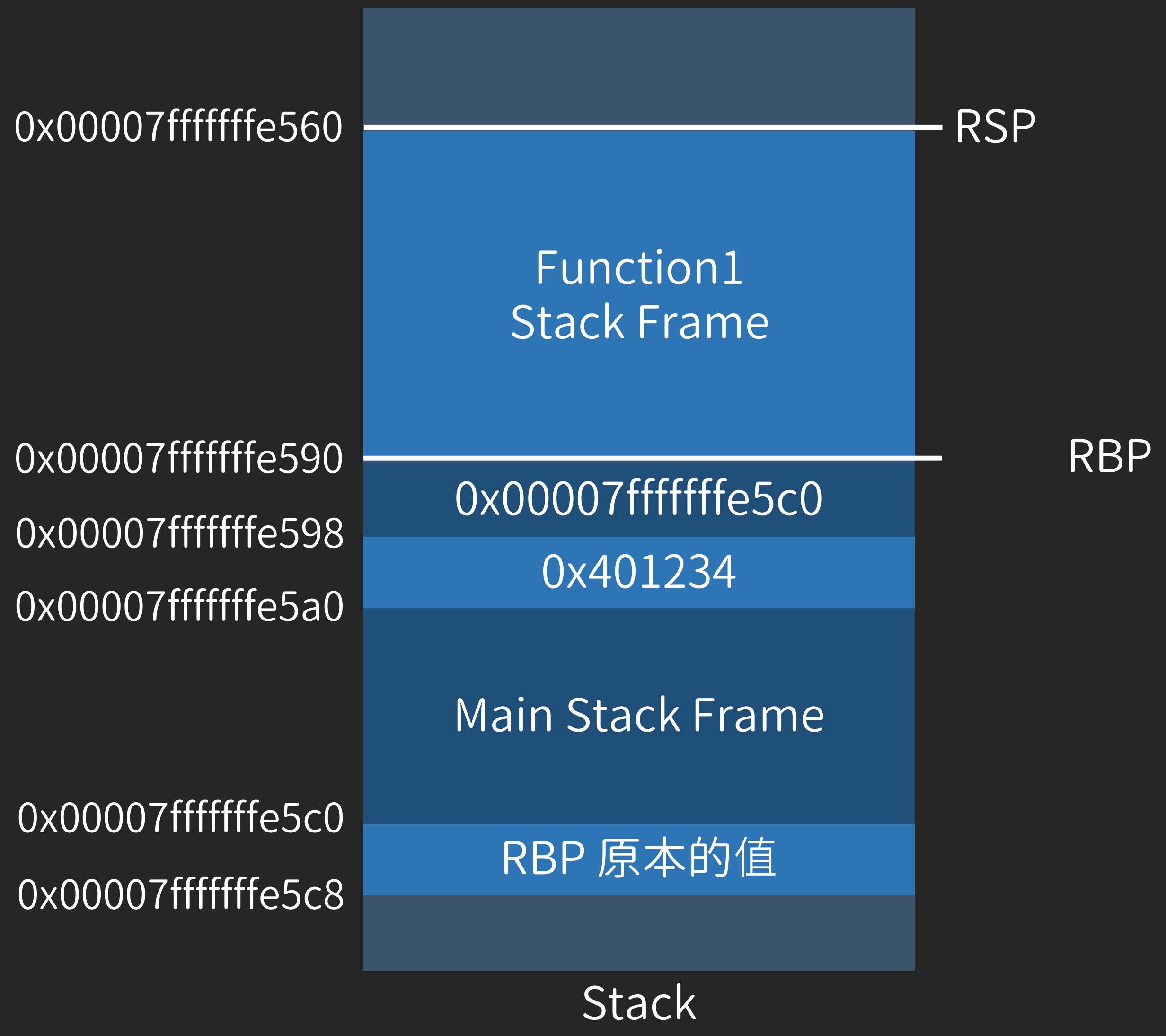
Stack

Stack Frame

```
         main
         push   rbp
         mov    rbp, rsp
         sub    rsp, 20h
         …
         call   function1
0x401234 leave
         ret
```

```
function1
push   rbp
mov    rbp, rsp
sub    rsp, 30h
…
leave
ret
```

```
leave
=
mov rsp, rbp
pop rbp
```



Stack Frame

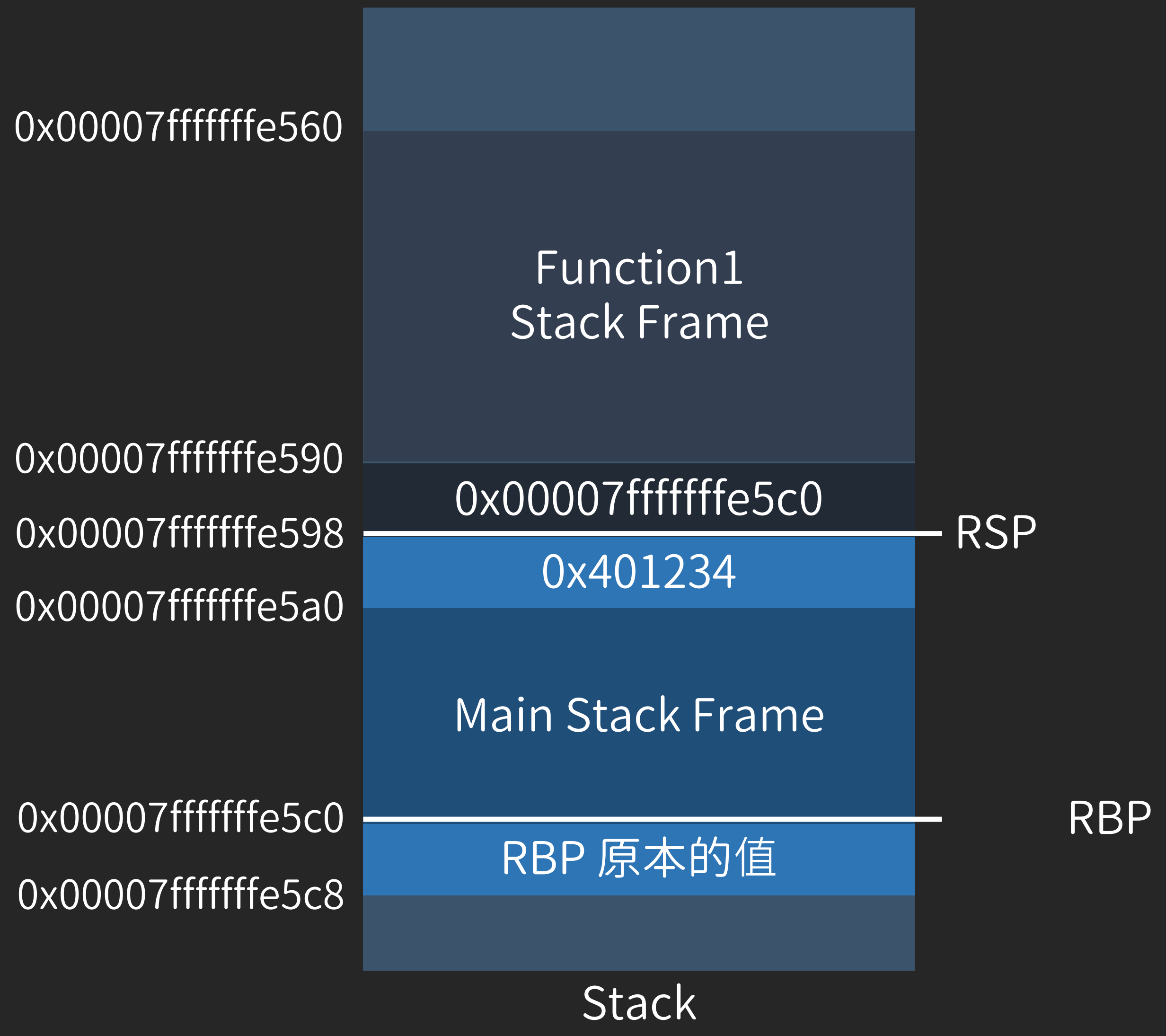
main

```
         push   rbp
         mov    rbp, rsp
         sub    rsp, 20h
         …
         call   function1
0x401234 leave
         ret
```

function1

```
push   rbp
mov    rbp, rsp
sub    rsp, 30h
…
leave
ret
```

```
leave
=
mov rsp, rbp
pop rbp
```



Stack Frame

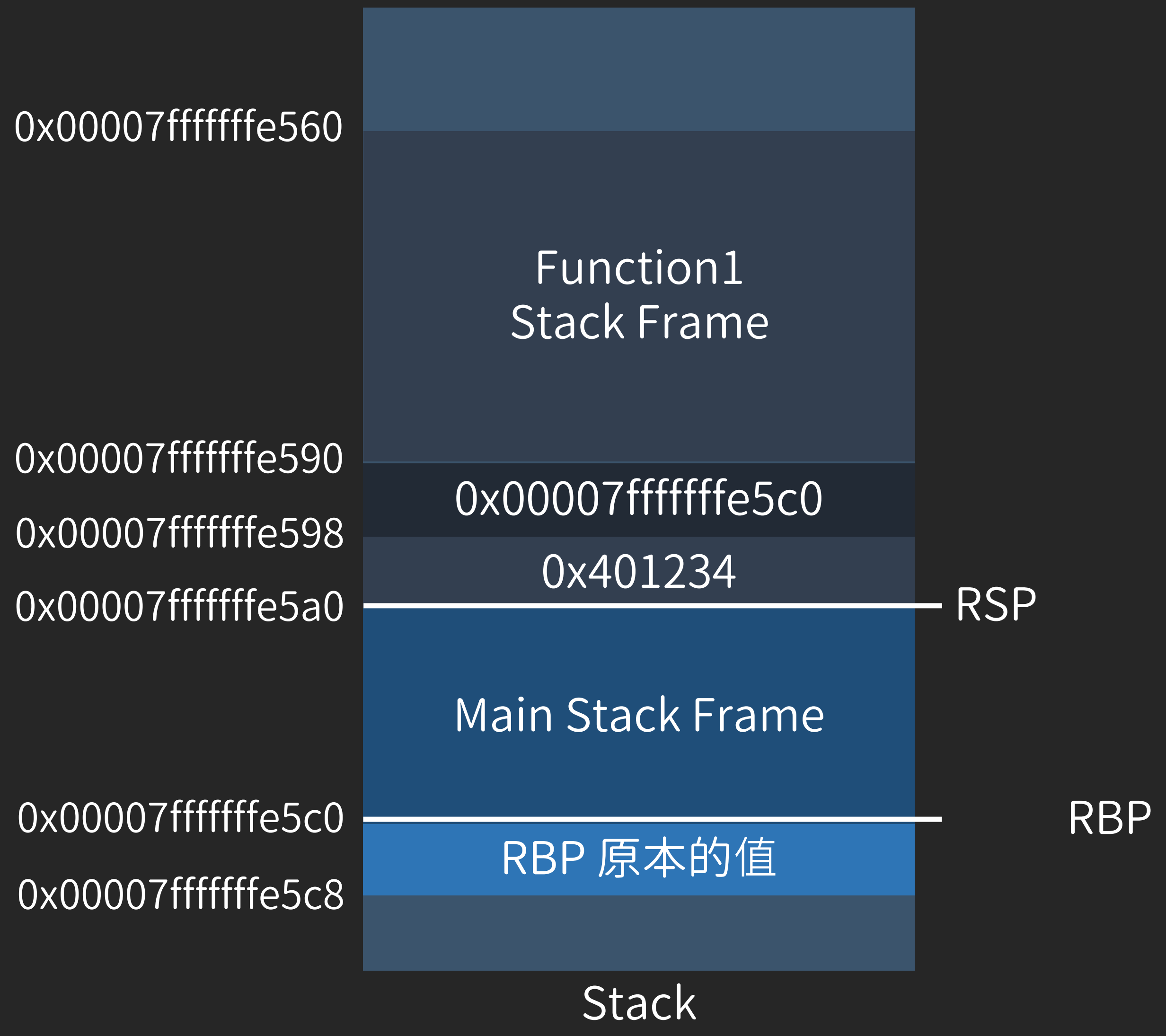
main

```
         push  rbp
         mov   rbp, rsp
         sub   rsp, 20h
         …
         call  function1
0x401234 leave
         ret
```

function1

```
push  rbp
mov   rbp, rsp
sub   rsp, 30h
…
leave
ret
```

```
leave
=
mov rsp, rbp
pop rbp
```



0x00007fffffffe560

Function1
Stack Frame

0x00007fffffffe590 | 0x00007fffffffe5c0

0x00007fffffffe598 | 0x401234

0x00007fffffffe5a0 — RSP

Main Stack Frame

0x00007fffffffe5c0 — RBP

0x00007fffffffe5c8 | RBP 原本的值

Stack

Stack Frame

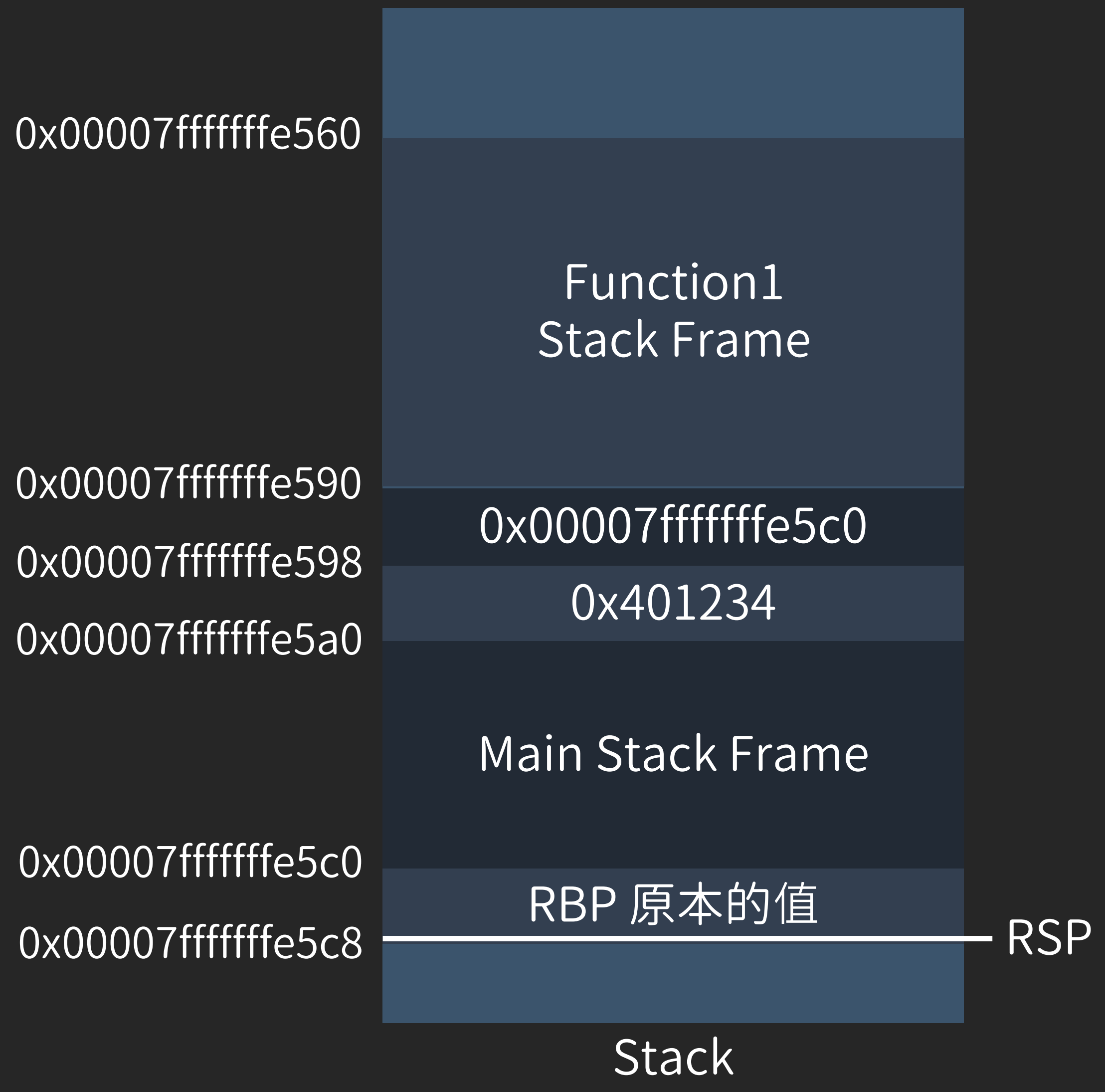
main

```
         push  rbp
         mov   rbp, rsp
         sub   rsp, 20h
         …
         call  function1
0x401234 leave
         ret
```

function1

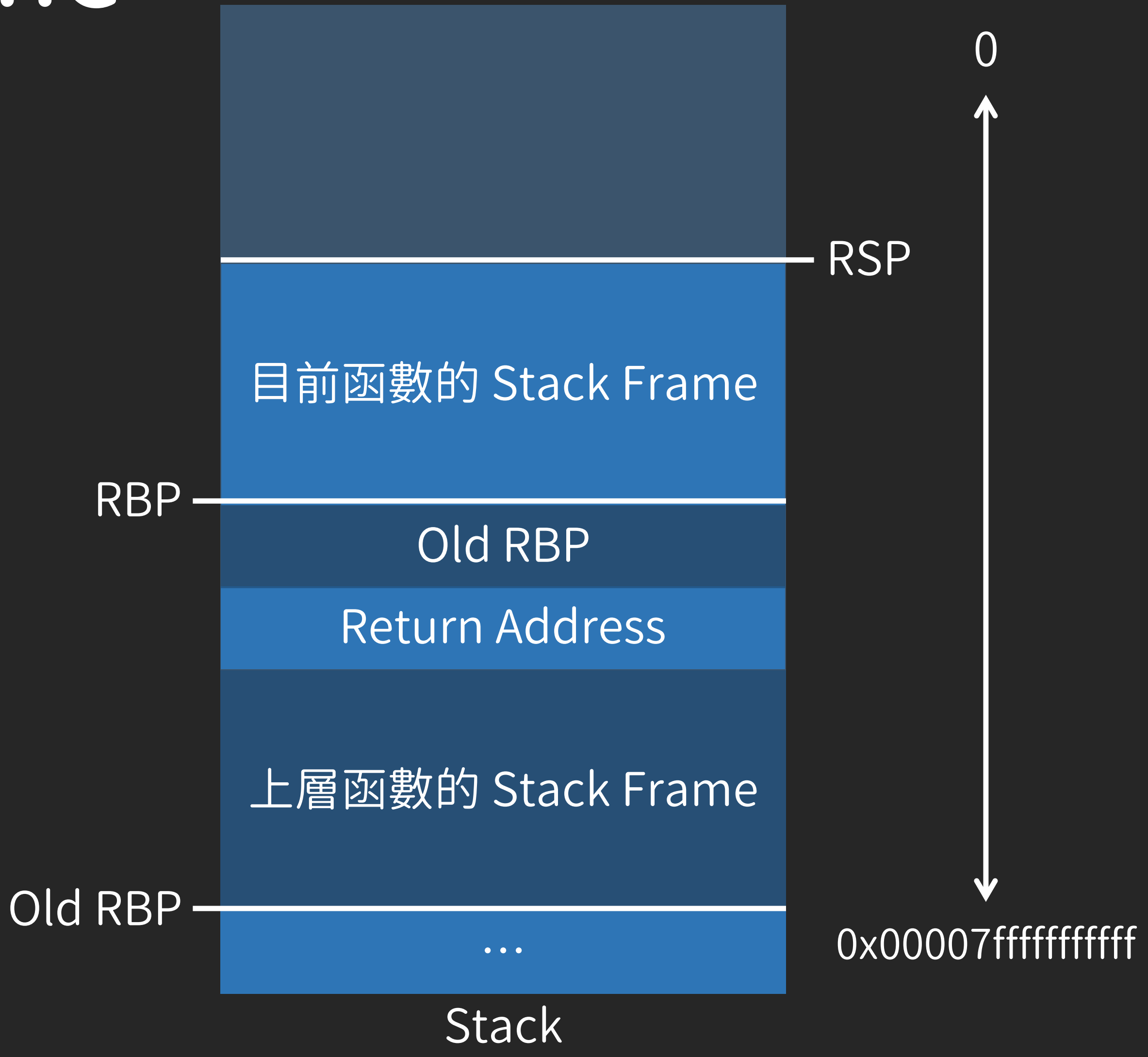
```
push  rbp
mov   rbp, rsp
sub   rsp, 30h
…
leave
ret
```

```
leave
=
mov rsp, rbp
pop rbp
```



Stack Frame

- 統整一下



Stack Frame

- Q1: 函數都是以 RSP 或 RBP 來定位區域變數，那怎麼區別不同函數的區域變數?
  - A1: 想辦法讓不同函數的 stack 區域不同

- Q2: 呼叫函數後, RIP 就從 A 函數跑到 B 函數了，要怎麼 return 回 A 函數?
  - A2: 在呼叫 B 函數前把下一條指令 push 進 stack
    B 函數執行 ret
    把 A 函數下一條指令從 stack pop 回 rip
    進而回到 A 函數

Struct

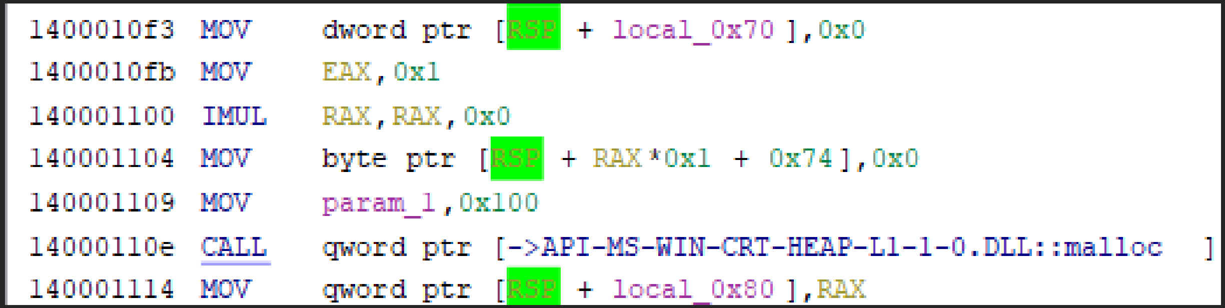
Struct

```c
#define NAME_SIZE 10
```

```c
typedef struct {
    int id;
    char name[NAME_SIZE];
    char *data;
} info_struct;
```

```c
info_struct local_info;
```

```c
local_info.id = 0;
local_info.name[0] = NULL;
local_info.data = malloc(0x100);
```



```
1400010f3  MOV     dword ptr [RSP + local_0x70 ],0x0
1400010fb  MOV     EAX,0x1
140001100  IMUL    RAX,RAX,0x0
140001104  MOV     byte ptr [RSP + RAX*0x1 + 0x74],0x0
140001109  MOV     param_1,0x100
14000110e  CALL    qword ptr [->API-MS-WIN-CRT-HEAP-L1-1-0.DLL::malloc  ]
140001114  MOV     qword ptr [RSP + local_0x80 ],RAX
```

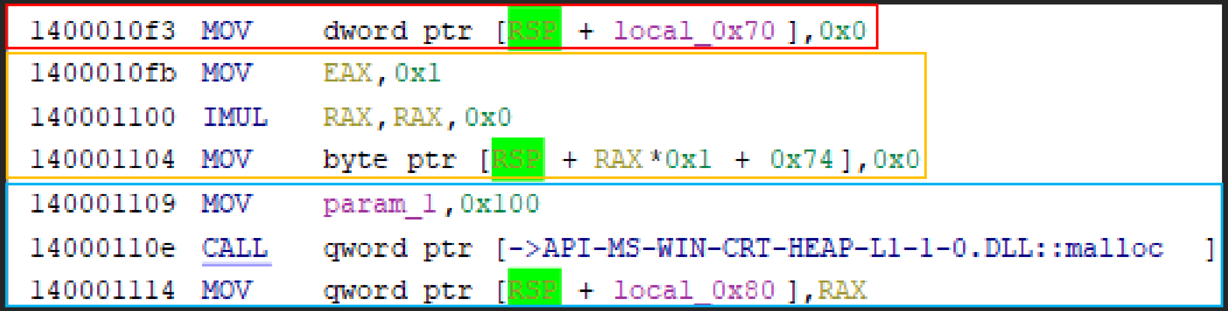
Struct

```c
#define NAME_SIZE 10
```

```c
typedef struct {
    int id;
    char name[NAME_SIZE];
    char *data;
} info_struct;
```

```c
info_struct local_info;
```

```c
local_info.id = 0;
local_info.name[0] = NULL;
local_info.data = malloc(0x100);
```



```
1400010f3  MOV     dword ptr [RSP + local_0x70 ],0x0
1400010fb  MOV     EAX,0x1
140001100  IMUL    RAX,RAX,0x0
140001104  MOV     byte ptr [RSP + RAX*0x1 + 0x74],0x0
140001109  MOV     param_1,0x100
14000110e  CALL    qword ptr [->API-MS-WIN-CRT-HEAP-L1-1-0.DLL::malloc  ]
140001114  MOV     qword ptr [RSP + local_0x80 ],RAX
```

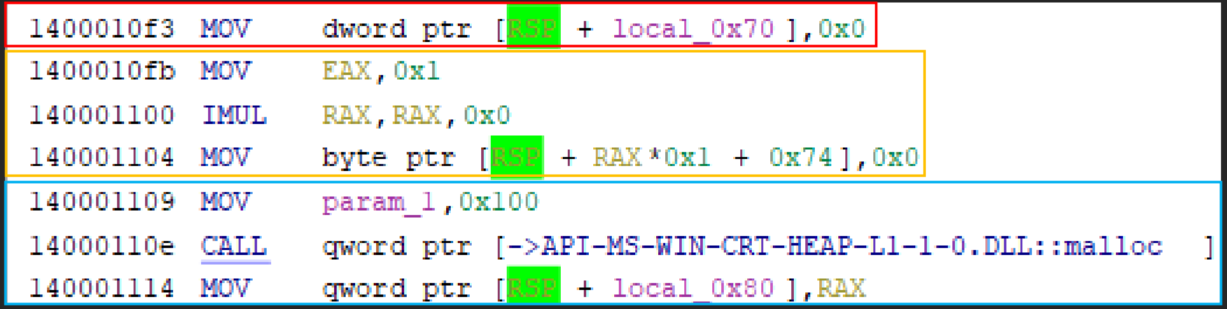
Struct

```c
#define NAME_SIZE 10
```

```c
typedef struct {
    int id;
    char name[NAME_SIZE];
    char *data;
} info_struct;
```

```c
info_struct local_info;
```

```c
local_info.id = 0;
local_info.name[0] = NULL;
local_info.data = malloc(0x100);
```

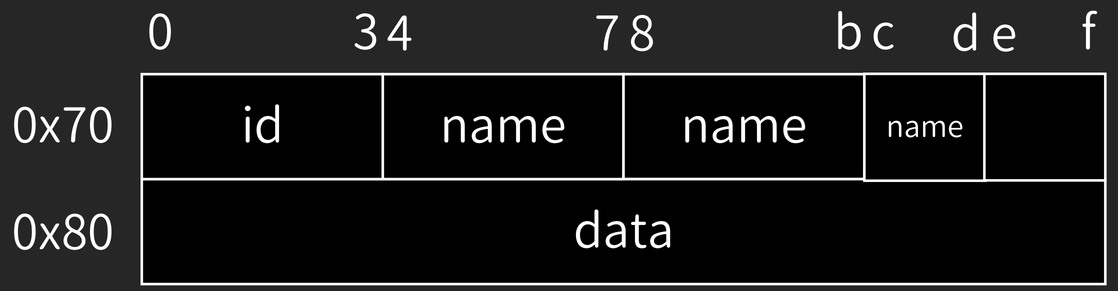


```
1400010f3  MOV     dword ptr [RSP + local_0x70 ],0x0
1400010fb  MOV     EAX,0x1
140001100  IMUL    RAX,RAX,0x0
140001104  MOV     byte ptr [RSP + RAX*0x1 + 0x74],0x0
140001109  MOV     param_1,0x100
14000110e  CALL    qword ptr [->API-MS-WIN-CRT-HEAP-L1-1-0.DLL::malloc   ]
140001114  MOV     qword ptr [RSP + local_0x80 ],RAX
```

可以觀察到 compiler 將 RSP + 0x70 的位址當 id

RSP + 0x74 的位址當 name 起始位址

將 RSP + 0x80 當 data



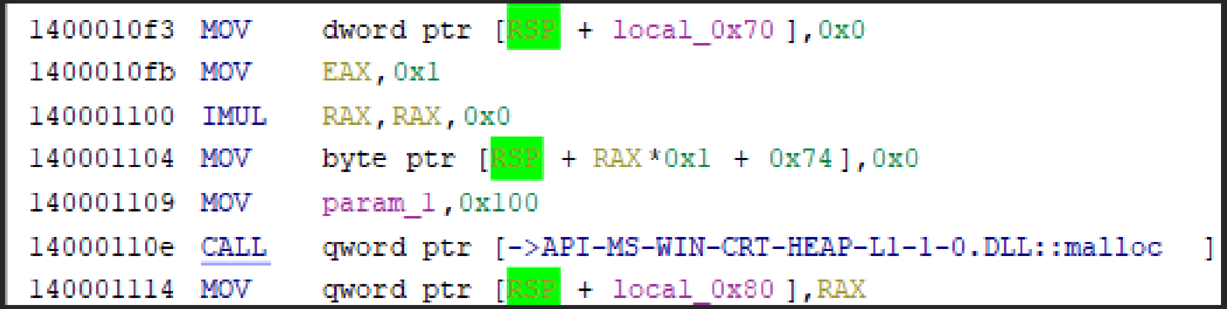
Struct

```
#define NAME_SIZE 10
```

```
typedef struct {
    int id;
    char name[NAME_SIZE];
    char *data;
} info_struct;
```

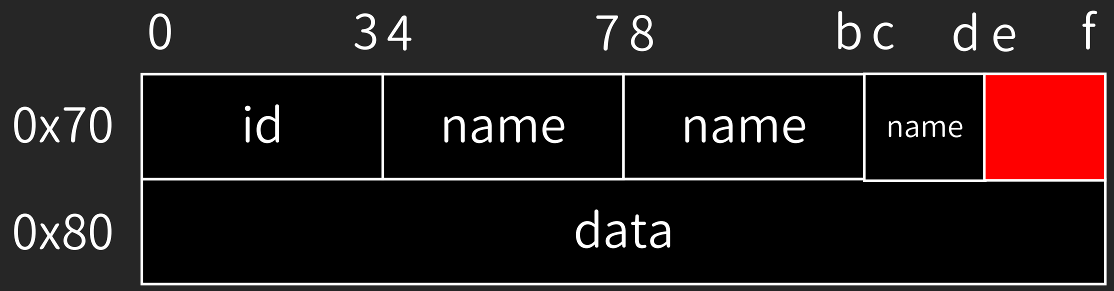
```
info_struct local_info;
```

```
local_info.id = 0;
local_info.name[0] = NULL;
local_info.data = malloc(0x100);
```



```
1400010f3  MOV     dword ptr [RSP + local_0x70 ],0x0
1400010fb  MOV     EAX,0x1
140001100  IMUL    RAX,RAX,0x0
140001104  MOV     byte ptr [RSP + RAX*0x1 + 0x74],0x0
140001109  MOV     param_1,0x100
14000110e  CALL    qword ptr [->API-MS-WIN-CRT-HEAP-L1-1-0.DLL::malloc  ]
140001114  MOV     qword ptr [RSP + local_0x80 ],RAX
```

有兩 Byte 沒有用到



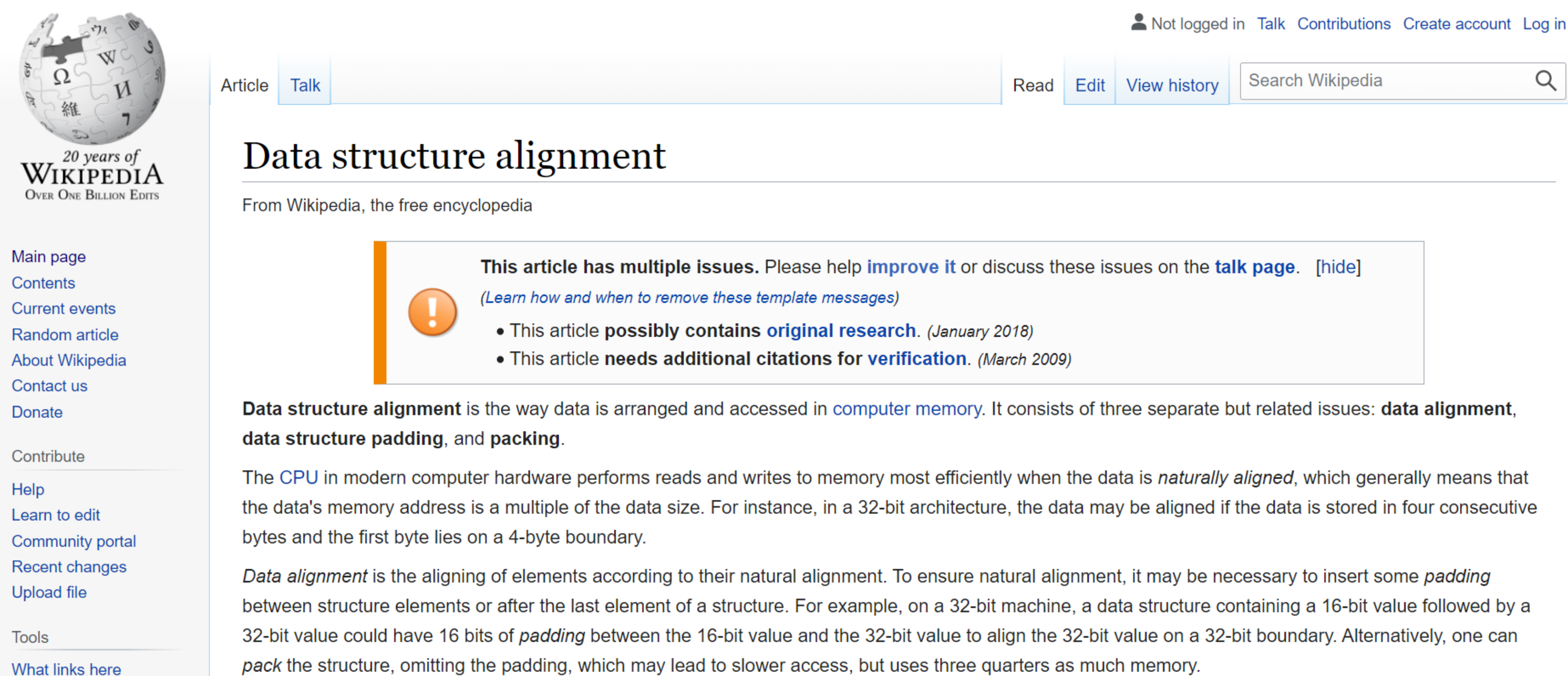
0 3 4 7 8 b c d e f

0x70 id name name name

0x80 data

Struct

- Struct alignment



Data structure alignment

From Wikipedia, the free encyclopedia

This article has multiple issues. Please help **improve it** or discuss these issues on the **talk page**. [hide]

(Learn how and when to remove these template messages)

- This article **possibly contains original research**. *(January 2018)*
- This article **needs additional citations for verification**. *(March 2009)*

Data structure alignment is the way data is arranged and accessed in computer memory. It consists of three separate but related issues: **data alignment**, **data structure padding**, and **packing**.

The CPU in modern computer hardware performs reads and writes to memory most efficiently when the data is *naturally aligned*, which generally means that the data's memory address is a multiple of the data size. For instance, in a 32-bit architecture, the data may be aligned if the data is stored in four consecutive bytes and the first byte lies on a 4-byte boundary.

Data alignment is the aligning of elements according to their natural alignment. To ensure natural alignment, it may be necessary to insert some *padding* between structure elements or after the last element of a structure. For example, on a 32-bit machine, a data structure containing a 16-bit value followed by a 32-bit value could have 16 bits of *padding* between the 16-bit value and the 32-bit value to align the 32-bit value on a 32-bit boundary. Alternatively, one can *pack* the structure, omitting the padding, which may lead to slower access, but uses three quarters as much memory.

Endian

Endian

- Byte 的順序

- 一個整數 0x12345678，兩種儲存方式

0	1	2	3
0x78	0x56	0x34	0x12

Little Endian

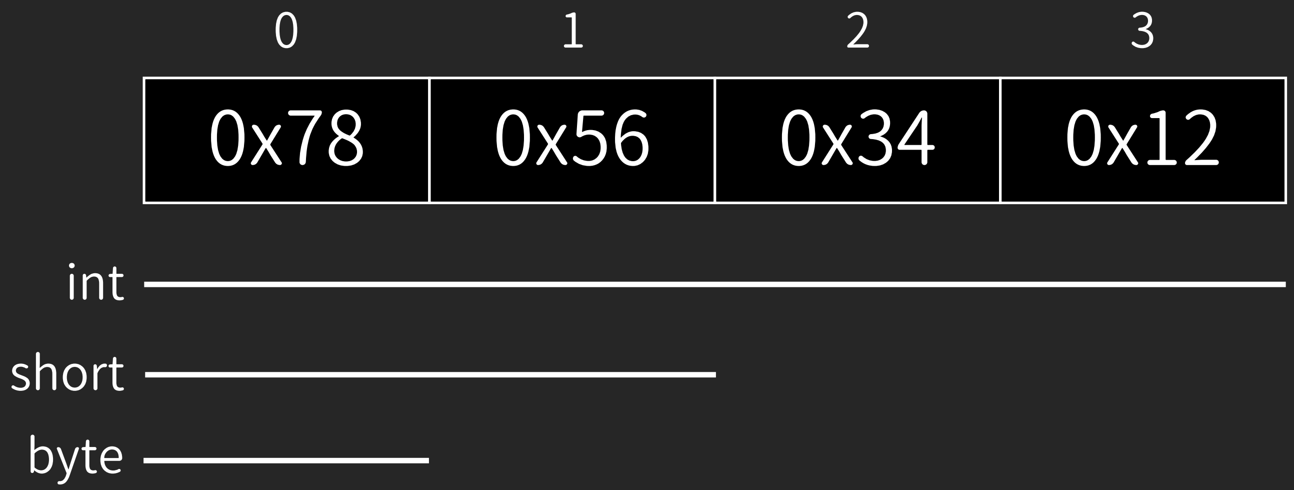
0	1	2	3
0x12	0x34	0x56	0x78

Big Endian

Endian

- 常見是用 Little Endian

- 將 int 0x12345678 轉成 short 0x5678, 起始位址不用改變



0	1	2	3
0x78	0x56	0x34	0x12

Little Endian

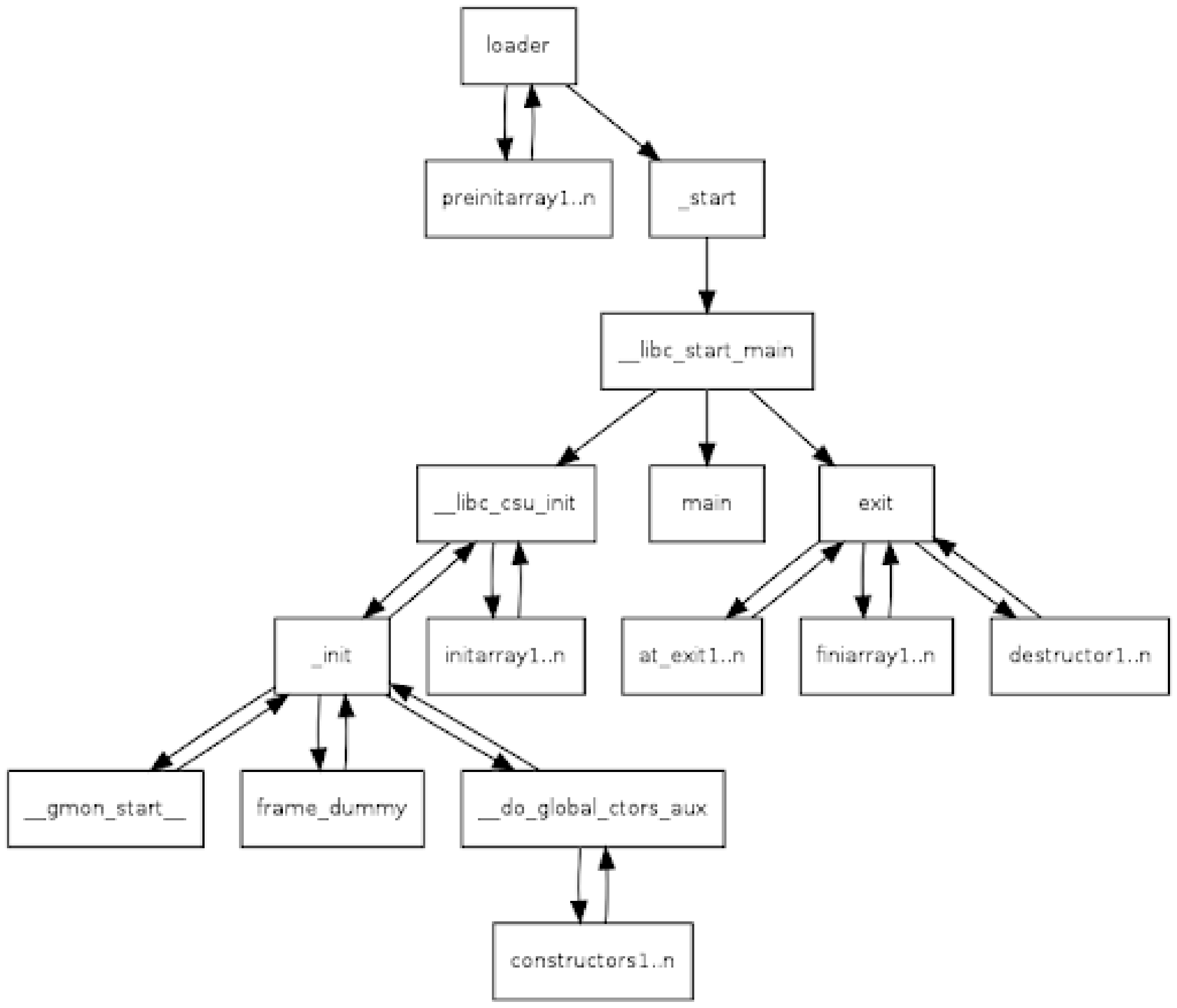
Lab 2

Where to start?

main ?

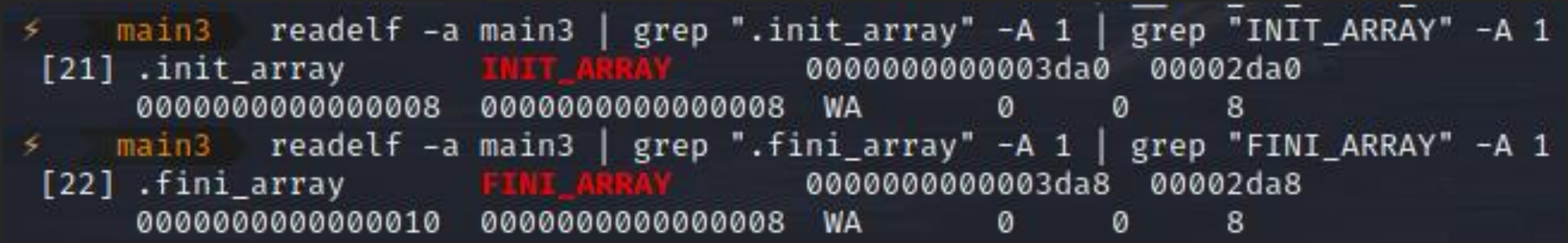
- 程式的第一條指令就是 main 嗎? 其實不是
- 其實有辦法讓某些程式碼比 main 還要早執行
- 想一下 C++, 全域物件的初始化是不是要比 main 還要早執行

__libc_start_main



INIT / FINI

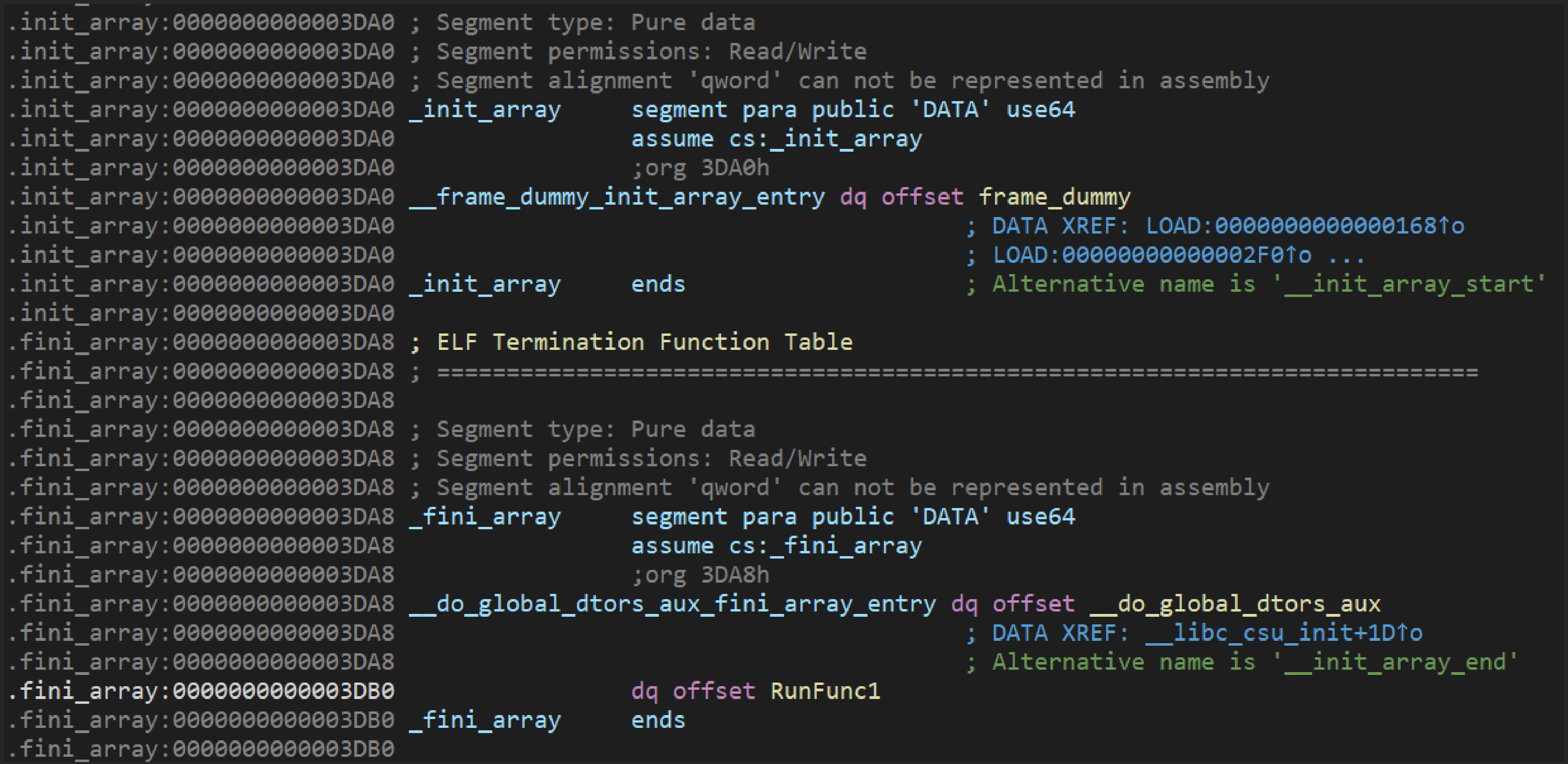
- 用 readelf 觀察一下



```
⚡  main3  readelf -a main3 | grep ".init_array" -A 1 | grep "INIT_ARRAY" -A 1
 [21] .init_array       INIT_ARRAY       0000000000003da0  00002da0
      0000000000000008  0000000000000008  WA       0     0     8
⚡  main3  readelf -a main3 | grep ".fini_array" -A 1 | grep "FINI_ARRAY" -A 1
 [22] .fini_array       FINI_ARRAY       0000000000003da8  00002da8
      0000000000000010  0000000000000008  WA       0     0     8
```

INIT / FINI

- 用 IDA 觀察一下



```
.init_array:0000000000003DA0 ; Segment type: Pure data
.init_array:0000000000003DA0 ; Segment permissions: Read/Write
.init_array:0000000000003DA0 ; Segment alignment 'qword' can not be represented in assembly
.init_array:0000000000003DA0 _init_array     segment para public 'DATA' use64
.init_array:0000000000003DA0                 assume cs:_init_array
.init_array:0000000000003DA0                 ;org 3DA0h
.init_array:0000000000003DA0 __frame_dummy_init_array_entry dq offset frame_dummy
.init_array:0000000000003DA0                                         ; DATA XREF: LOAD:0000000000000168↑o
.init_array:0000000000003DA0                                         ; LOAD:00000000000002F0↑o ...
.init_array:0000000000003DA0 _init_array     ends                    ; Alternative name is '__init_array_start'
.init_array:0000000000003DA0
.fini_array:0000000000003DA8 ; ELF Termination Function Table
.fini_array:0000000000003DA8 ; ===========================================================================
.fini_array:0000000000003DA8
.fini_array:0000000000003DA8 ; Segment type: Pure data
.fini_array:0000000000003DA8 ; Segment permissions: Read/Write
.fini_array:0000000000003DA8 ; Segment alignment 'qword' can not be represented in assembly
.fini_array:0000000000003DA8 _fini_array     segment para public 'DATA' use64
.fini_array:0000000000003DA8                 assume cs:_fini_array
.fini_array:0000000000003DA8                 ;org 3DA8h
.fini_array:0000000000003DA8 __do_global_dtors_aux_fini_array_entry dq offset __do_global_dtors_aux
.fini_array:0000000000003DA8                                         ; DATA XREF: __libc_csu_init+1D↑o
.fini_array:0000000000003DA8                                         ; Alternative name is '__init_array_end'
.fini_array:0000000000003DB0                 dq offset RunFunc1
.fini_array:0000000000003DB0 _fini_array     ends
.fini_array:0000000000003DB0
```

Where to start?

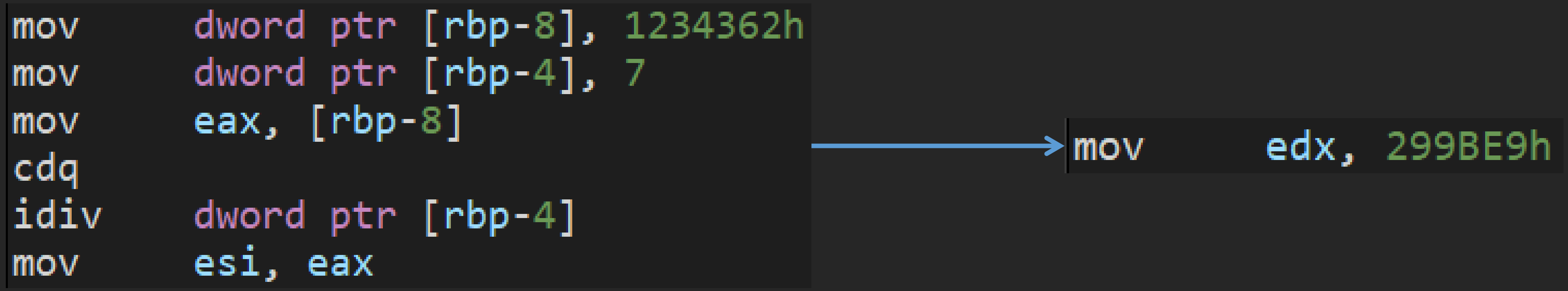
- Init / fini 函數指針放在一個 array 中
- 並會在初始 / 結束階段呼叫到
- .init_array / .fini_array
- 有機會藏 code 的地方, 需另外注意一下

Compiler Optimization

Compiler Optimization

- Compiler 想方設法優化編出來的 code, 使其跑更快

- e.g. 能先跑完算完的 code 直接算完



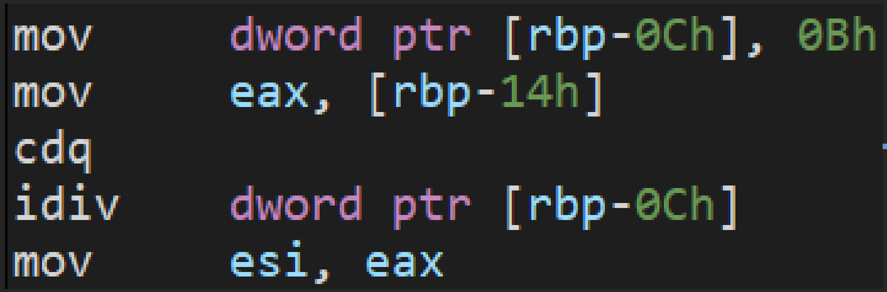
```
mov     dword ptr [rbp-8], 1234362h
mov     dword ptr [rbp-4], 7
mov     eax, [rbp-8]
cdq
idiv    dword ptr [rbp-4]
mov     esi, eax
```

```
mov     edx, 299BE9h
```

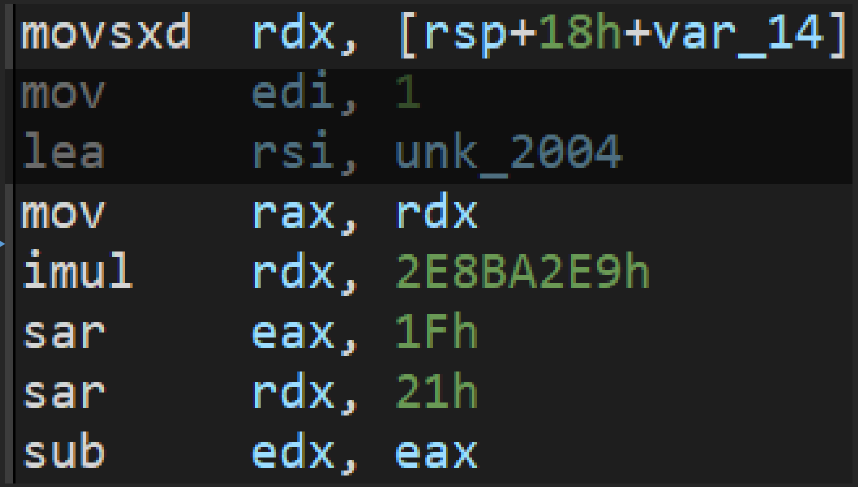
Compiler Optimization

- idiv 很慢, 右邊雖然指令數較多但還比較快 (較暗區段為無關的指令)

- 所以右邊那坨是什麼鬼 = =



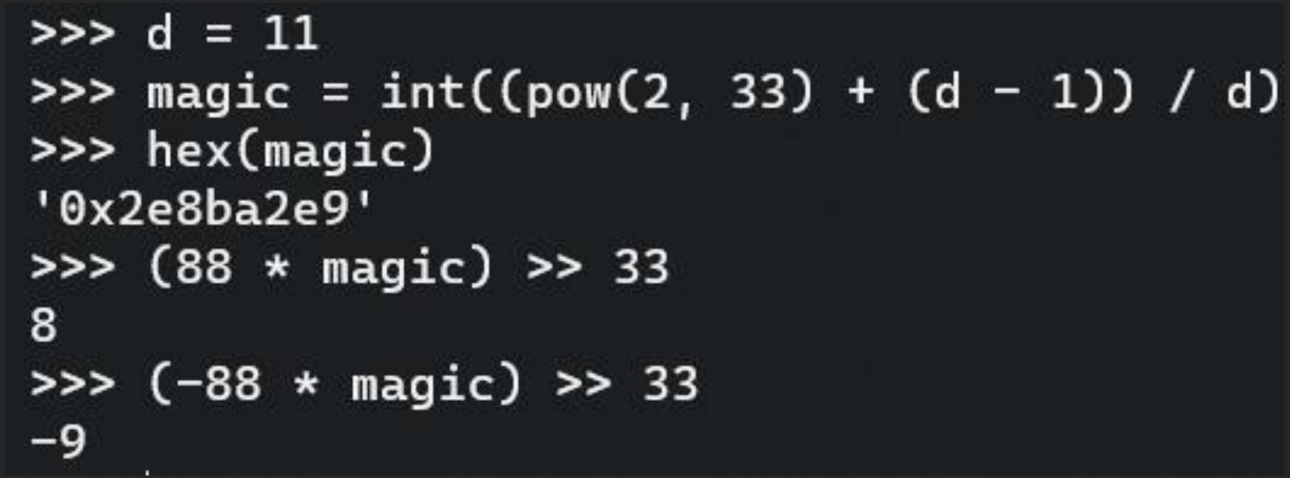
```
mov     dword ptr [rbp-0Ch], 0Bh
mov     eax, [rbp-14h]
cdq
idiv    dword ptr [rbp-0Ch]
mov     esi, eax
```



```
movsxd  rdx, [rsp+18h+var_14]
mov     edi, 1
lea     rsi, unk_2004
mov     rax, rdx
imul    rdx, 2E8BA2E9h
sar     eax, 1Fh
sar     rdx, 21h
sub     edx, eax
```

Compiler Optimization

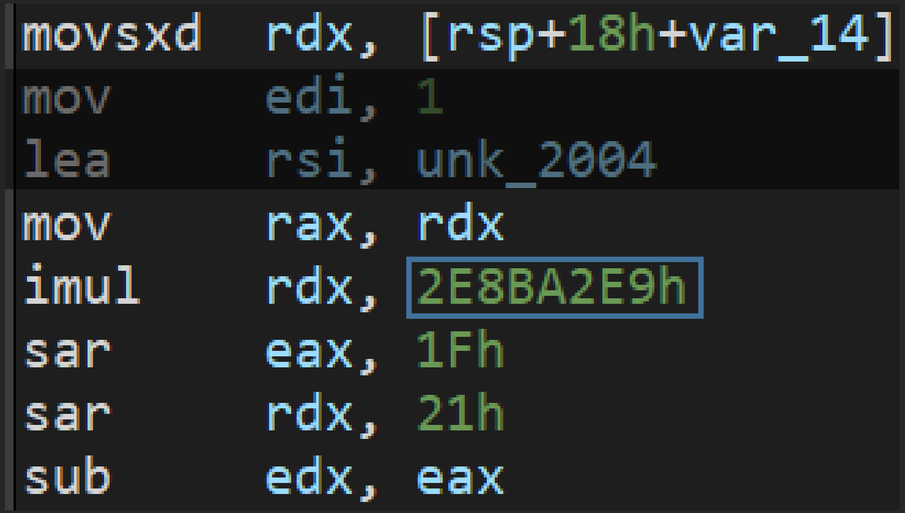
- 原本除法改成 $\frac{a}{11} = \frac{2^{33}}{11}\frac{a}{2^{33}}$
- 可以先算完 $\frac{2^{33}}{11}$ (取 ceil)
- 乘完 a 後, 用右移完成 $\frac{a}{2^{33}}$
- a 若是負數, 則需加一



```
>>> d = 11
>>> magic = int((pow(2, 33) + (d - 1)) / d)
>>> hex(magic)
'0x2e8ba2e9'
>>> (88 * magic) >> 33
8
>>> (-88 * magic) >> 33
-9
```

Compiler Optimization

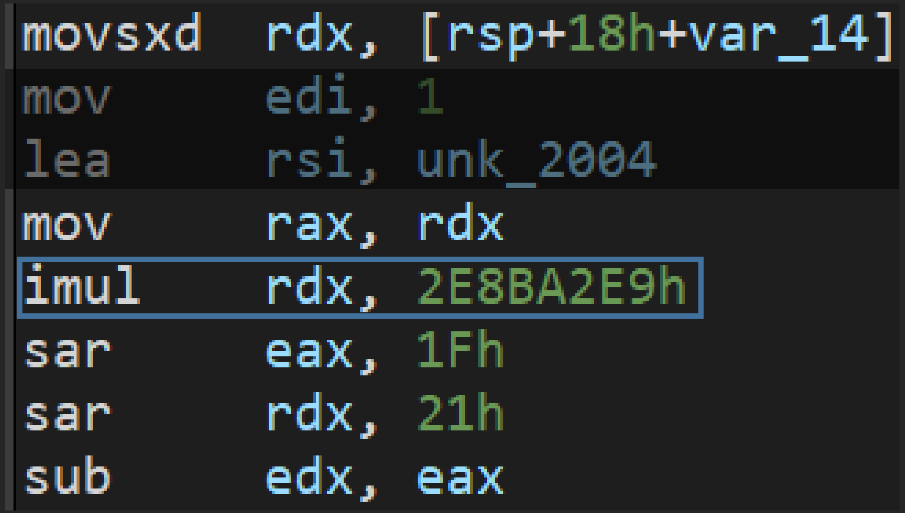
- 原本除法改成 $\frac{a}{11} = \frac{2^{33}}{11} \frac{a}{2^{33}}$
- 可以先算完 $\frac{2^{33}}{11}$ (取 ceil)
- 乘完 a 後, 用右移完成 $\frac{a}{2^{33}}$
- a 若是負數, 則需加一



```
movsxd  rdx, [rsp+18h+var_14]
mov     edi, 1
lea     rsi, unk_2004
mov     rax, rdx
imul    rdx, 2E8BA2E9h
sar     eax, 1Fh
sar     rdx, 21h
sub     edx, eax
```

Compiler Optimization

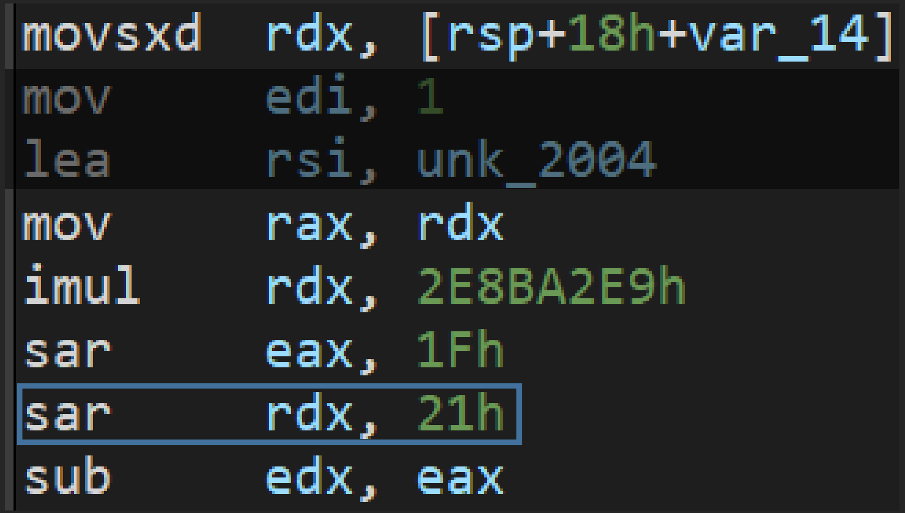
- 原本除法改成 $\frac{a}{11} = \frac{2^{33}}{11} \frac{a}{2^{33}}$
- 可以先算完 $\frac{2^{33}}{11}$ (取 ceil)
- 乘完 a 後, 用右移完成 $\frac{a}{2^{33}}$
- a 若是負數, 則需加一



```
movsxd  rdx, [rsp+18h+var_14]
mov     edi, 1
lea     rsi, unk_2004
mov     rax, rdx
imul    rdx, 2E8BA2E9h
sar     eax, 1Fh
sar     rdx, 21h
sub     edx, eax
```

Compiler Optimization

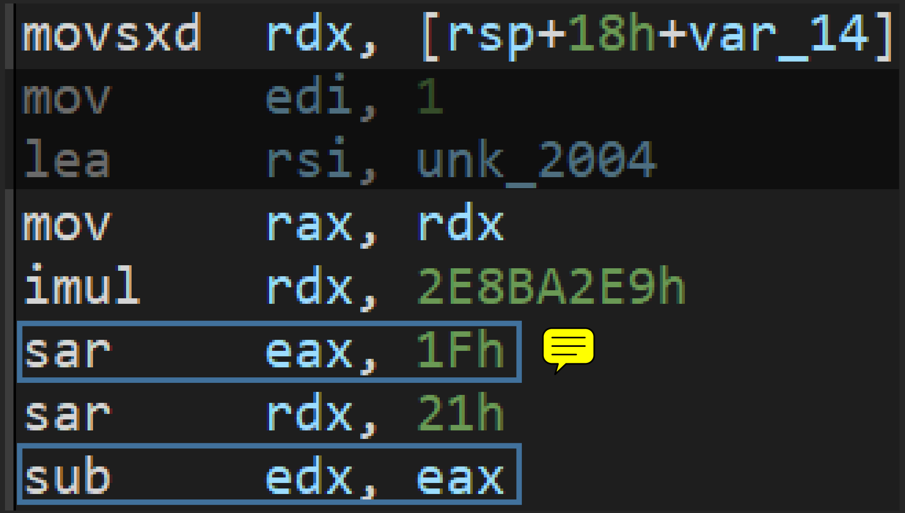
- 原本除法改成 $\frac{a}{11} = \frac{2^{33}}{11} \frac{a}{2^{33}}$
- 可以先算完 $\frac{2^{33}}{11}$ (取 ceil)
- 乘完 a 後, 用右移完成 $\frac{a}{2^{33}}$
- a 若是負數, 則需加一



```
movsxd  rdx, [rsp+18h+var_14]
mov     edi, 1
lea     rsi, unk_2004
mov     rax, rdx
imul    rdx, 2E8BA2E9h
sar     eax, 1Fh
sar     rdx, 21h
sub     edx, eax
```

Compiler Optimization

- 原本除法改成 $\frac{a}{11} = \frac{2^{33}}{11} \frac{a}{2^{33}}$
- 可以先算完 $\frac{2^{33}}{11}$ (取 ceil)
- 乘完 a 後, 用右移完成 $\frac{a}{2^{33}}$
- a 若是負數, 則需加一



```
movsxd  rdx, [rsp+18h+var_14]
mov     edi, 1
lea     rsi, unk_2004
mov     rax, rdx
imul    rdx, 2E8BA2E9h
sar     eax, 1Fh
sar     rdx, 21h
sub     edx, eax
```

ASLR

ASLR

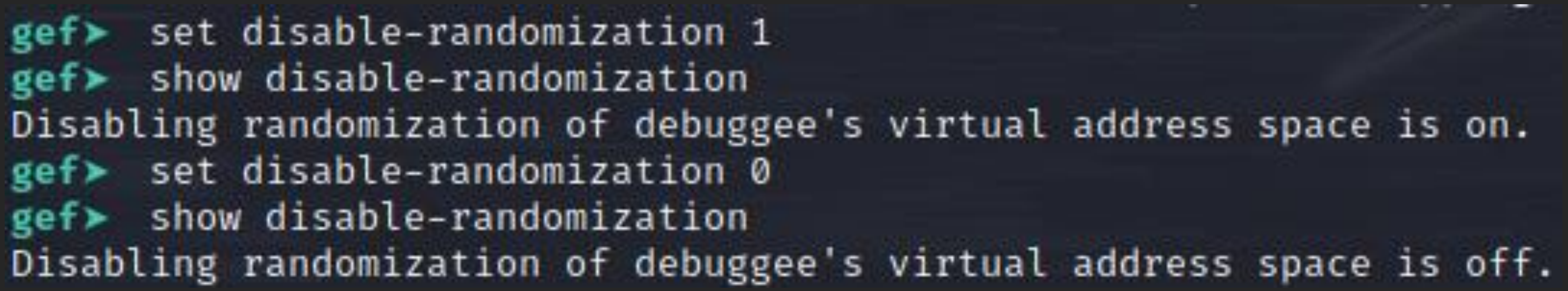
- Address Space Layout Randomization

- 使 library 的 base address 是隨機的

- 使得漏洞利用更加困難

ASLR

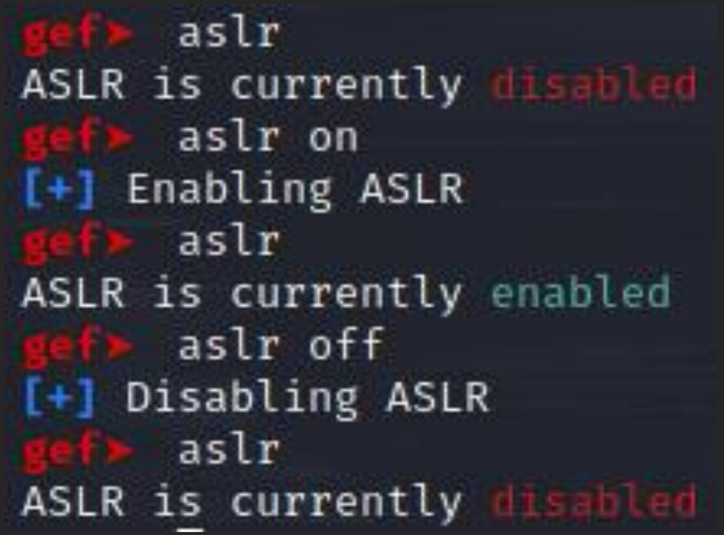
- 在 gdb 中可以用指令關閉 ASLR 後再 debug
- set/show disable-randomization



```
gef➤  set disable-randomization 1
gef➤  show disable-randomization
Disabling randomization of debuggee's virtual address space is on.
gef➤  set disable-randomization 0
gef➤  show disable-randomization
Disabling randomization of debuggee's virtual address space is off.
```

ASLR

- gef 簡化成 aslr 指令

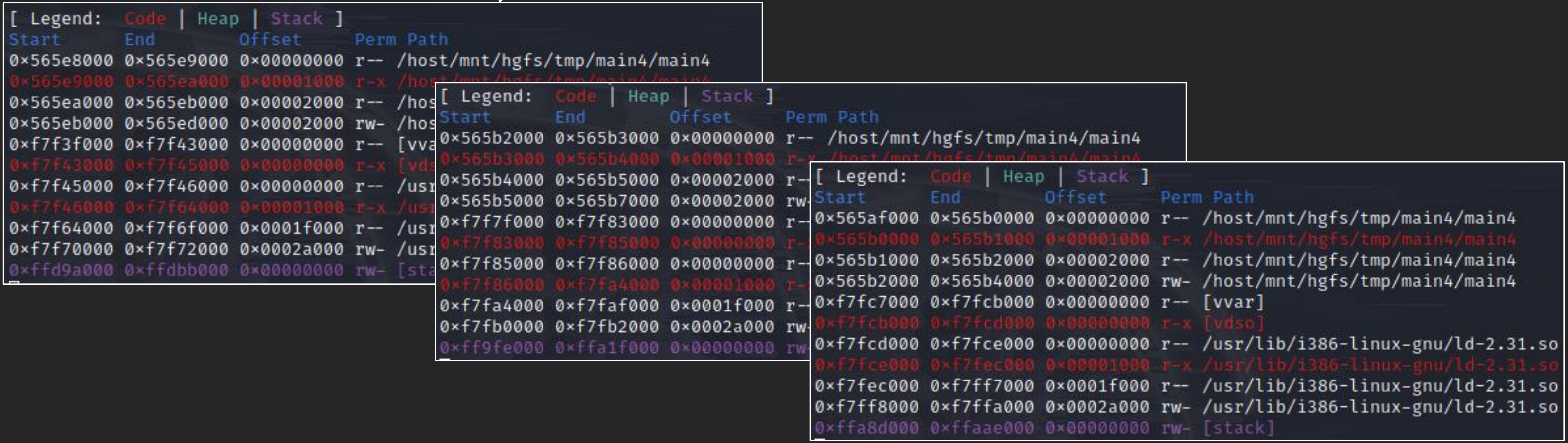


```
gef➤  aslr
ASLR is currently disabled
gef➤  aslr on
[+] Enabling ASLR
gef➤  aslr
ASLR is currently enabled
gef➤  aslr off
[+] Disabling ASLR
gef➤  aslr
ASLR is currently disabled
```

ASLR

- 把 ASLR 啟用後, 觀察記憶體空間

- 會發現每次執行時, 記憶體位址都不太一樣



```
[ Legend:  Code | Heap | Stack ]
Start      End        Offset     Perm Path
0x565e8000 0x565e9000 0x00000000 r-- /host/mnt/hgfs/tmp/main4/main4
0x565e9000 0x565ea000 0x00001000 r-x /host/mnt/hgfs/tmp/main4/main4
0x565ea000 0x565eb000 0x00002000 r-- /hos
0x565eb000 0x565ed000 0x00002000 rw- /hos
0xf7f3f000 0xf7f43000 0x00000000 r-- [vva
0xf7f43000 0xf7f45000 0x00000000 r-x [vds
0xf7f45000 0xf7f46000 0x00000000 r-- /usr
0xf7f46000 0xf7f64000 0x00001000 r-x /usr
0xf7f64000 0xf7f6f000 0x0001f000 r-- /usr
0xf7f70000 0xf7f72000 0x0002a000 rw- /usr
0xffd9a000 0xffdbb000 0x00000000 rw- [sta
```

```
[ Legend:  Code | Heap | Stack ]
Start      End        Offset     Perm Path
0x565b2000 0x565b3000 0x00000000 r-- /host/mnt/hgfs/tmp/main4/main4
0x565b3000 0x565b4000 0x00001000 r-x /host/mnt/hgfs/tmp/main4/main4
0x565b4000 0x565b5000 0x00002000 r--
0x565b5000 0x565b7000 0x00002000 rw-
0xf7f7f000 0xf7f83000 0x00000000 r--
0xf7f83000 0xf7f85000 0x00000000 r-
0xf7f85000 0xf7f86000 0x00000000 r--
0xf7f86000 0xf7fa4000 0x00001000 r-
0xf7fa4000 0xf7faf000 0x0001f000 r--
0xf7fb0000 0xf7fb2000 0x0002a000 rw-
0xff9fe000 0xffa1f000 0x00000000 rw
```

```
[ Legend:  Code | Heap | Stack ]
Start      End        Offset     Perm Path
0x565af000 0x565b0000 0x00000000 r-- /host/mnt/hgfs/tmp/main4/main4
0x565b0000 0x565b1000 0x00001000 r-x /host/mnt/hgfs/tmp/main4/main4
0x565b1000 0x565b2000 0x00002000 r-- /host/mnt/hgfs/tmp/main4/main4
0x565b2000 0x565b4000 0x00002000 rw- /host/mnt/hgfs/tmp/main4/main4
0xf7fc7000 0xf7fcb000 0x00000000 r-- [vvar]
0xf7fcb000 0xf7fcd000 0x00000000 r-x [vdso]
0xf7fcd000 0xf7fce000 0x00000000 r-- /usr/lib/i386-linux-gnu/ld-2.31.so
0xf7fce000 0xf7fec000 0x00001000 r-x /usr/lib/i386-linux-gnu/ld-2.31.so
0xf7fec000 0xf7ff7000 0x0001f000 r-- /usr/lib/i386-linux-gnu/ld-2.31.so
0xf7ff8000 0xf7ffa000 0x0002a000 rw- /usr/lib/i386-linux-gnu/ld-2.31.so
0xffa8d000 0xffaae000 0x00000000 rw- [stack]
```

ASLR

- Linux 怎麼關 ASLR?

- System-wide
  - /proc/sys/kernel/randomize_va_space
  - sudo sysctl kernel.randomize_va_space=0

- Non-system-wide
  - Syscall personality: ADDR_NO_RANDOMIZE

Lab 3

Q & A

下課囉 \(. _ .)>