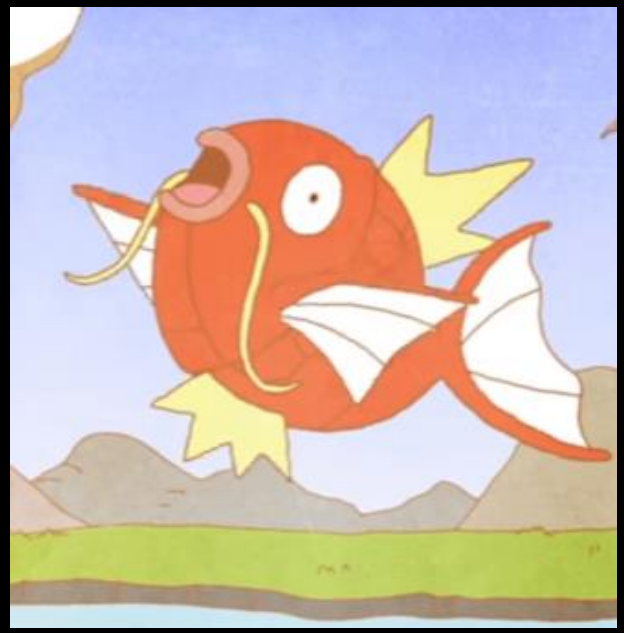
ROP

Return Oriented Programming

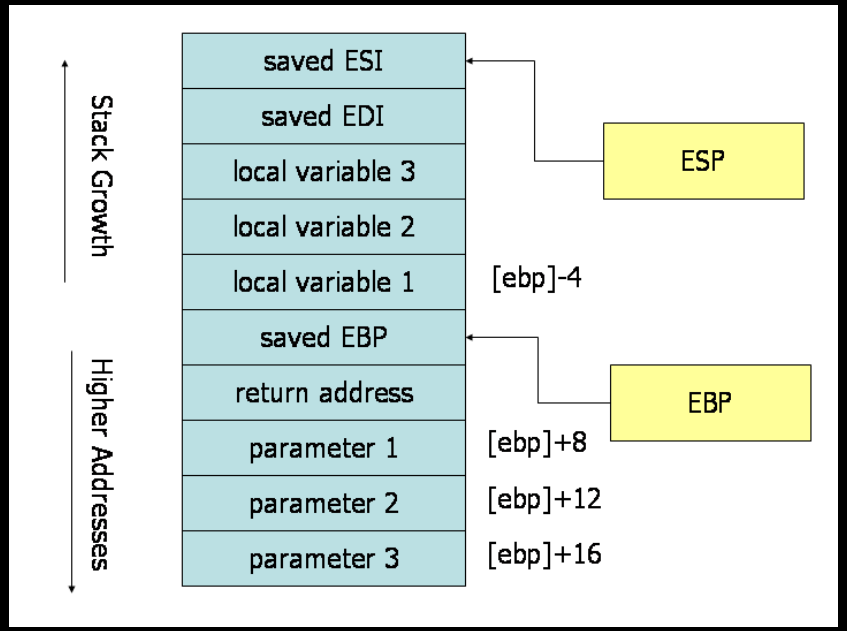
whoami

- briansp8210
- 交通大學資工系大二
- 喜歡學習 pwn 題的相關技巧 XD



Review

- When a program use some dangerous input method like gets() or scanf("%s", buf) …, buffer overflow may happen !
- By x86 calling convention, return address is stored at EBP+4, padding proper length then we can control the program !

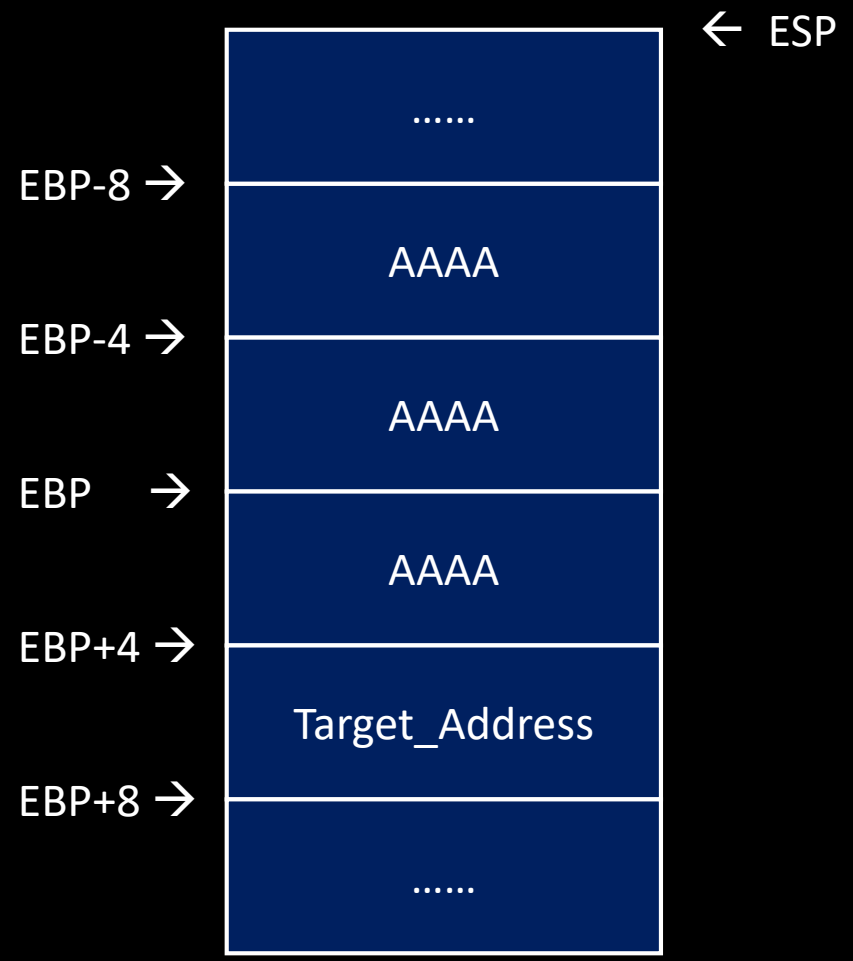


reference

After buffer overflow

Function Epilogue :

```
mov   esp, ebp
pop   ebp
ret
```

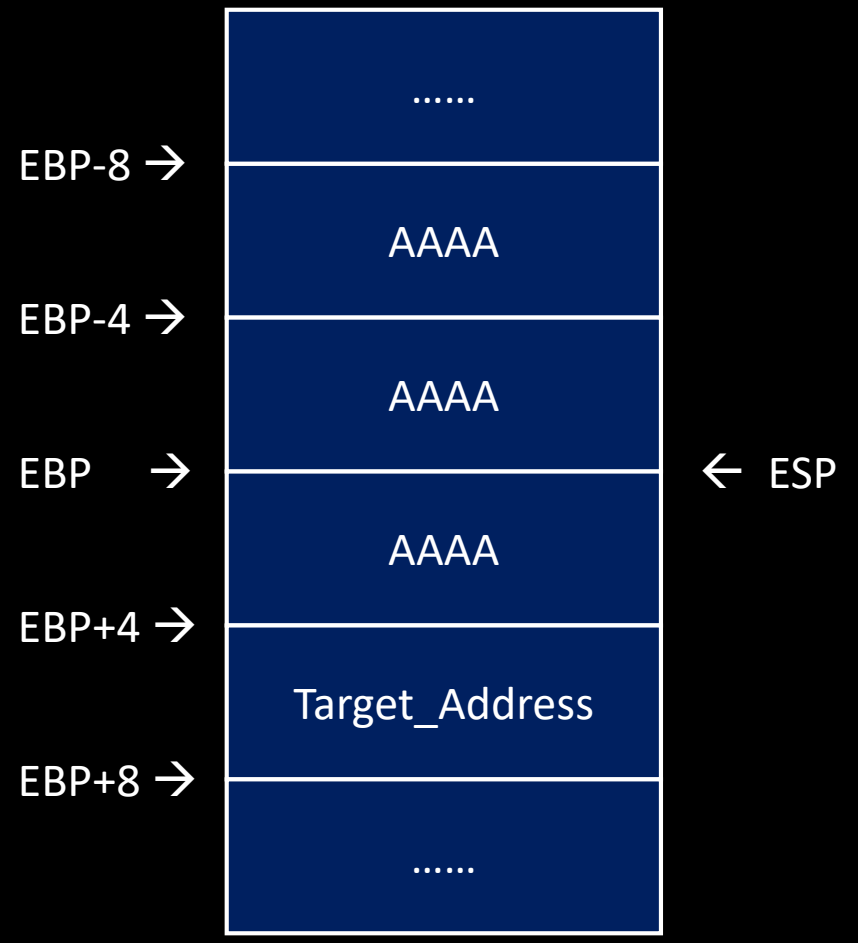


這份投影片裡用黃字標示的指令代表剛跑完這條指令，準備去跑下一條了。

After buffer overflow

Function Epilogue :

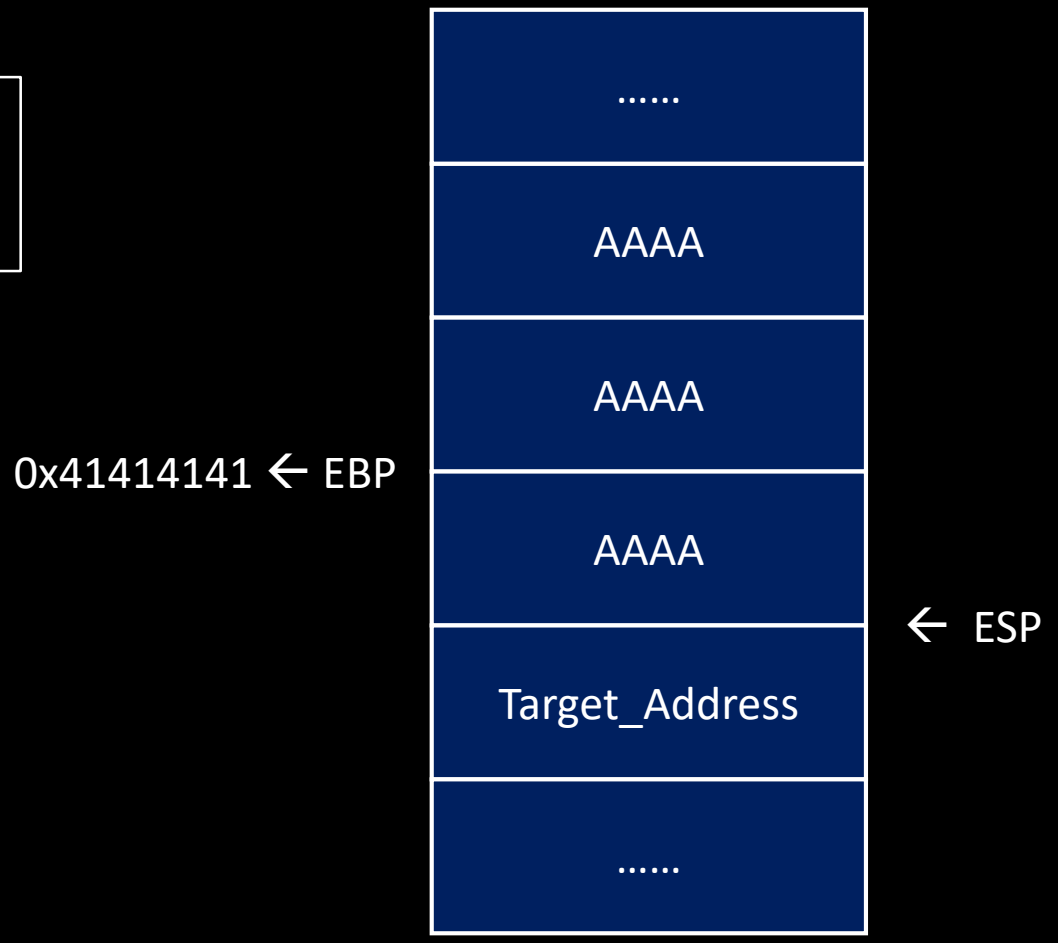
```
mov  esp, ebp
pop  ebp
ret
```



After buffer overflow

Function Epilogue :

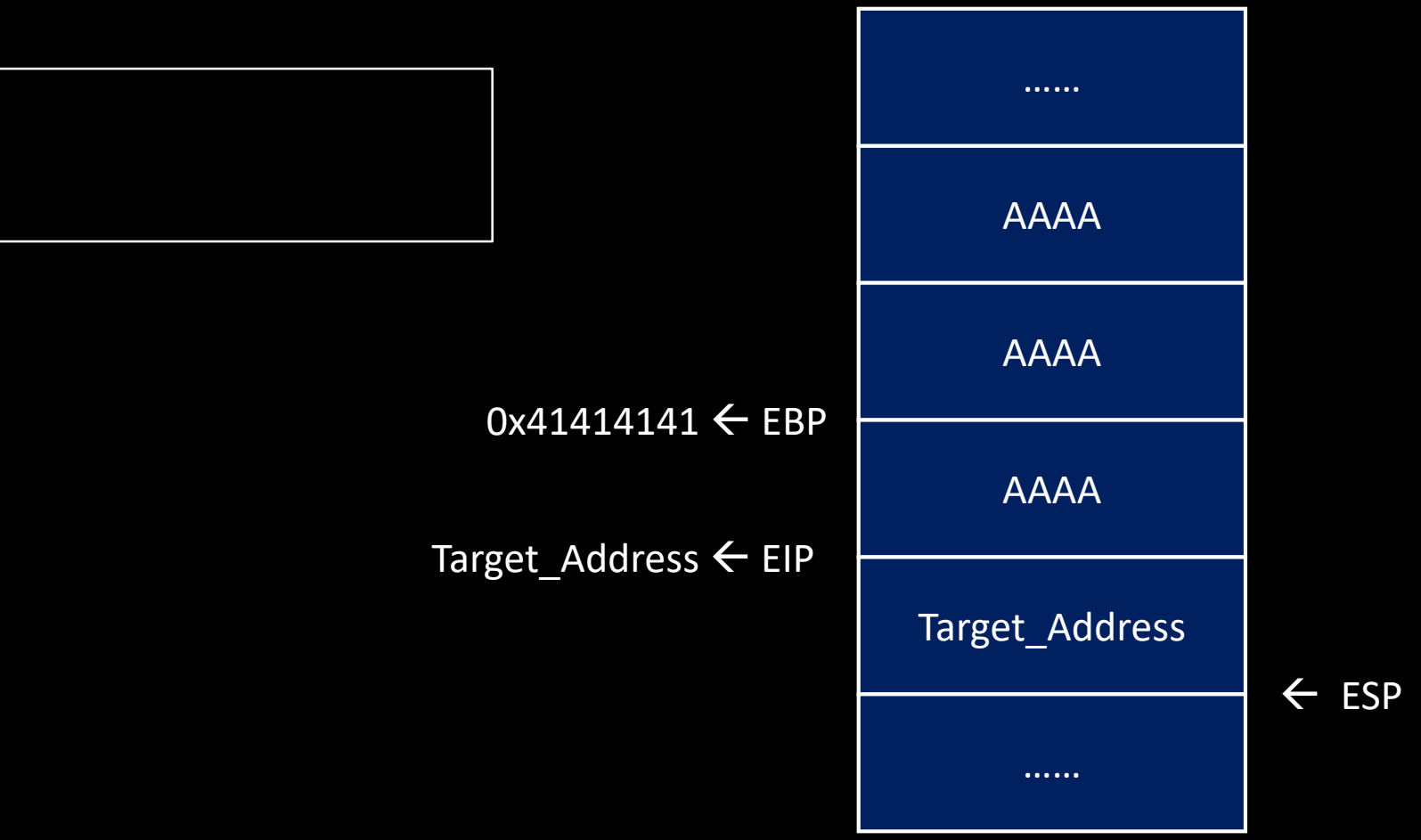
```
mov  esp, ebp
pop  ebp
ret
```



After buffer overflow

Function Epilogue :

```
mov  esp, ebp
pop  ebp
ret
```



What is the next step ?

- ret2text
- Return to shellcode
- Return to libc
- ROP

What is the next step ?

- ret2text
- Return to shellcode
- ret2libc
- ROP

ret2text

- When PIE (Position Independent Executable) is enable, program will be loaded to random address, we can’t predict where to return.

```
080485fd <secure>:
 80485fd:       55                      push   ebp
 80485fe:       89 e5                   mov    ebp,esp
 8048600:       83 ec 28                sub    esp,0x28
 8048603:       c7 04 24 00 00 00 00    mov    DWORD PTR [esp],0x0
 804860a:       e8 61 fe ff ff          call   8048470 <time@plt>
 804860f:       89 04 24                mov    DWORD PTR [esp],eax
 8048612:       e8 99 fe ff ff          call   80484b0 <srand@plt>
 8048617:       e8 c4 fe ff ff          call   80484e0 <rand@plt>
 804861c:       89 45 f4                mov    DWORD PTR [ebp-0xc],eax
```

```
>>> print ELF('ret2text').checksec()
RELRO:    Partial RELRO
Stack:    No canary found
NX:       NX enabled
PIE:      No PIE
```

```
000007eb <secure>:
 7eb:   55                      push   ebp
 7ec:   89 e5                   mov    ebp,esp
 7ee:   53                      push   ebx
 7ef:   83 ec 24                sub    esp,0x24
 7f2:   e8 c9 fe ff ff          call   6c0 <__x86.get_pc_thunk.bx>
 7f7:   81 c3 09 18 00 00       add    ebx,0x1809
 7fd:   c7 04 24 00 00 00 00    mov    DWORD PTR [esp],0x0
 804:   e8 d7 fd ff ff          call   5e0 <time@plt>
 809:   89 04 24                mov    DWORD PTR [esp],eax
 80c:   e8 1f fe ff ff          call   630 <srand@plt>
 811:   e8 4a fe ff ff          call   660 <rand@plt>
 816:   89 45 f4                mov    DWORD PTR [ebp-0xc],eax
```

```
>>> print ELF('ret2text_with_PIE').checksec()
RELRO:    Partial RELRO
Stack:    No canary found
NX:       NX enabled
PIE:      PIE enabled
```

- If not, we can use desirable program code to do exploit !

ret2text

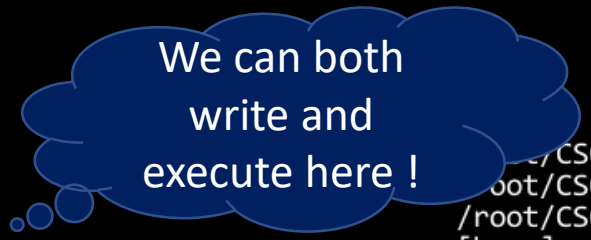
```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void secure(void)
{
    int secretcode, input;
    srand(time(NULL));

    secretcode = rand();
    scanf("%d", &input);
    if(input == secretcode)
        system("/bin/sh");
}

int main(void)
{
    setvbuf(stdout, 0LL, 2, 0LL);
    setvbuf(stdin, 0LL, 1, 0LL);

    char buf[100];

    printf("There is something amazing here, do you know anything?\n");
    gets(buf);
    printf("Maybe I will tell you next time !");

    return 0;
}
```

What is the next step ?

- ret2text
- Return to shellcode
- ret2libc
- ROP

Return to shellcode

- When memory address isn't marked as non-executable , we can send shellcode to it and return to its address to run our shellcode.



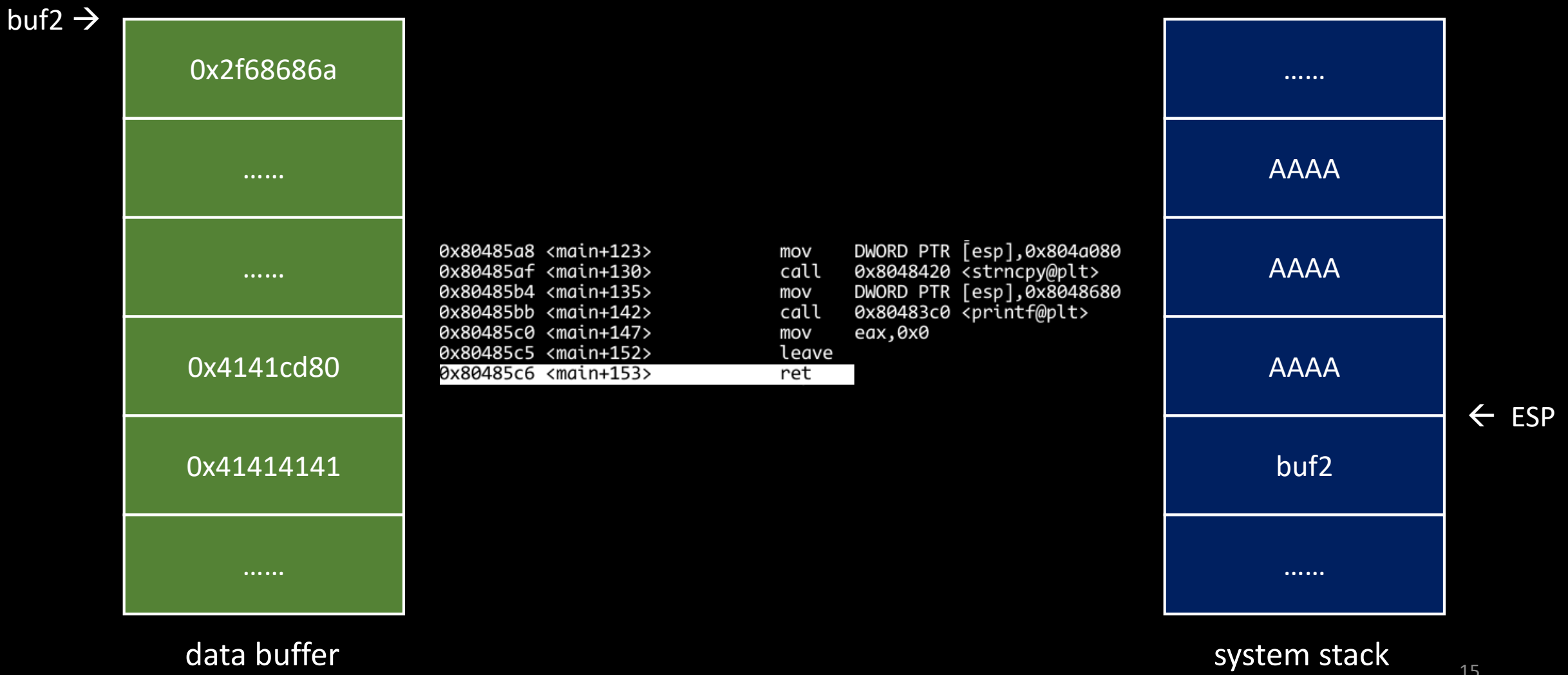

```
08048000-08049000 r-xp 00000000 08:01 928493      /CSC/ROP/return_to_shellcode/return_to_shellcode
08049000-0804a000 r-xp 00000000 08:01 928493      oot/CSC/ROP/return_to_shellcode/return_to_shellcode
0804a000-0804b000 rwxp 00001000 08:01 928493      /root/CSC/ROP/return_to_shellcode/return_to_shellcode
09df9000-09e1a000 rwxp 00000000 00:00 0           [heap]
f7599000-f759a000 rwxp 00000000 00:00 0
f759a000-f7746000 r-xp 00000000 08:01 3014705     /lib32/libc-2.23.so
f7746000-f7747000 ---p 001ac000 08:01 3014705     /lib32/libc-2.23.so
f7747000-f7749000 r-xp 001ac000 08:01 3014705     /lib32/libc-2.23.so
f7749000-f774a000 rwxp 001ae000 08:01 3014705     /lib32/libc-2.23.so
f774a000-f774e000 rwxp 00000000 00:00 0
f776d000-f776f000 r--p 00000000 00:00 0           [vvar]
f776f000-f7770000 r-xp 00000000 00:00 0           [vdso]
f7770000-f7792000 r-xp 00000000 08:01 3014701     /lib32/ld-2.23.so
f7792000-f7793000 rwxp 00000000 00:00 0
f7793000-f7794000 r-xp 00022000 08:01 3014701     /lib32/ld-2.23.so
f7794000-f7795000 rwxp 00023000 08:01 3014701     /lib32/ld-2.23.so
ffd96000-ffdb7000 rwxp 00000000 00:00 0           [stack]
```

```
>>> print ELF('return_to_shellcode').checksec()
RELRO:    Partial RELRO
Stack:    No canary found
NX:       NX disabled
PIE:      No PIE
```

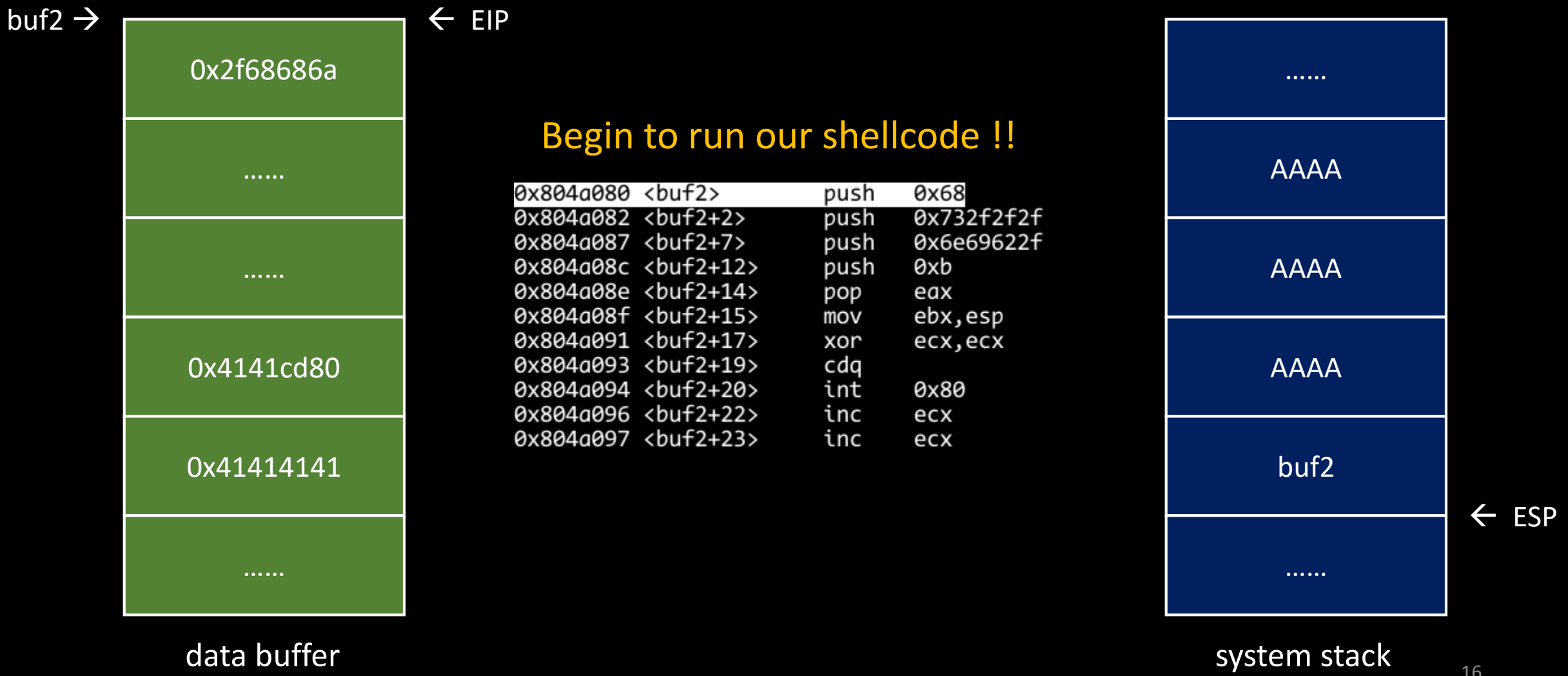
Return to shellcode

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

char buf2[100];

int main(void)
{
    setvbuf(stdout, 0LL, 2, 0LL);
    setvbuf(stdin, 0LL, 1, 0LL);

    char buf[100];

    printf("No system for you this time !!!\n");
    gets(buf);
    strncpy(buf2, buf, 100);
    printf("bye bye ~");

    return 0;
}
```

Return to shellcode



Return to shellcode



What is the next step ?

- ret2text
- Return to shellcode
- ret2libc
- ROP

Call a function

- Observe normal function call
  `strcpy(Destination,Source);`

```
.....
mov    DWORD PTR [esp+0x4],0x804a020
mov    DWORD PTR [esp],0x804a035
call   80482f0 <strcpy@plt>
.....
```

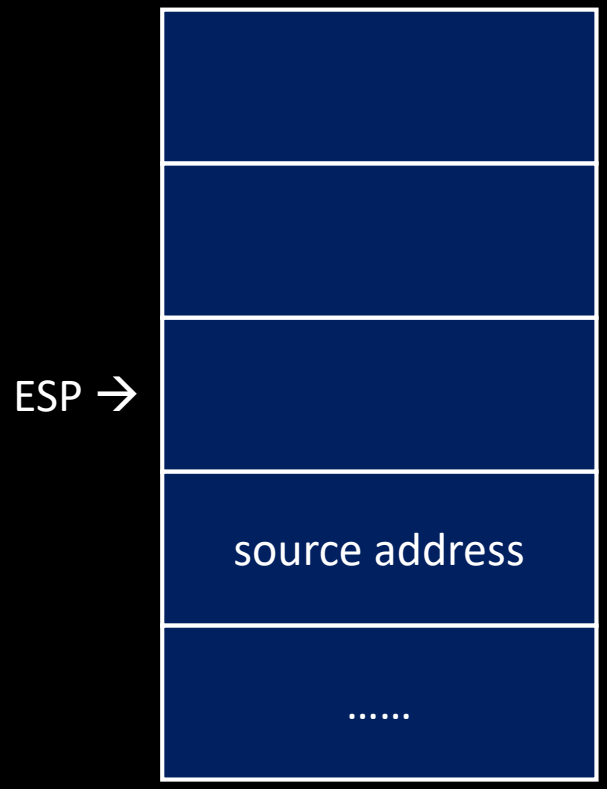
Source address: 0x804a020
Destination address: 0x804a035

Call a function

- Observe normal function call
  `strcpy(Destination,Source);`

```
.....
mov    DWORD PTR [esp+0x4],0x804a020
mov    DWORD PTR [esp],0x804a035
call   80482f0 <strcpy@plt>
.....
```

Source address: 0x804a020
Destination address: 0x804a035

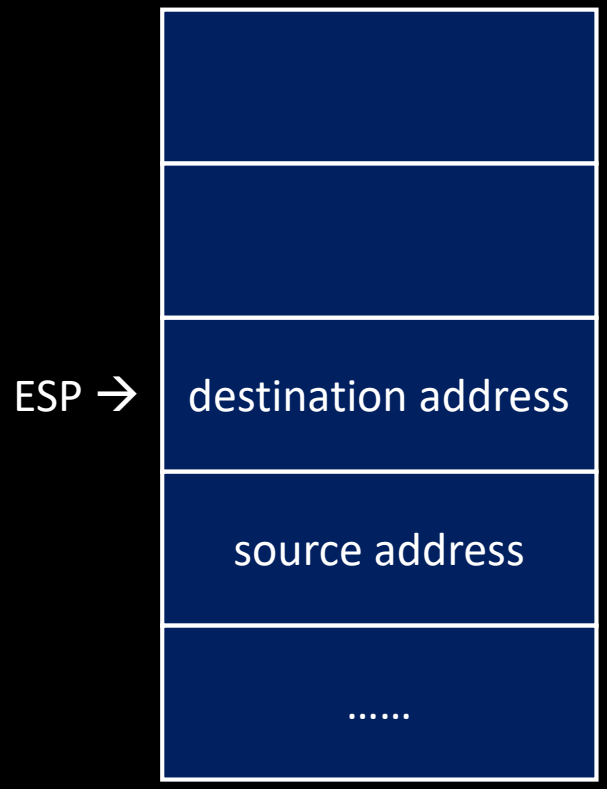


Call a function

- Observe normal function call
  `strcpy(Destination,Source);`

```
.....
mov     DWORD PTR [esp+0x4],0x804a020
mov     DWORD PTR [esp],0x804a035
call    80482f0 <strcpy@plt>
.....
```

Source address: 0x804a020
Destination address: 0x804a035

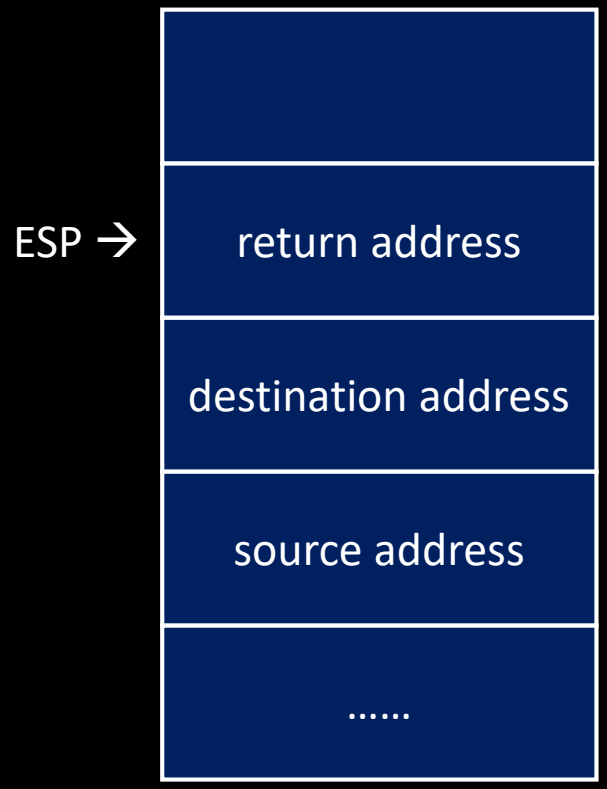


Call a function

- Observe normal function call
  `strcpy(Destination,Source);`

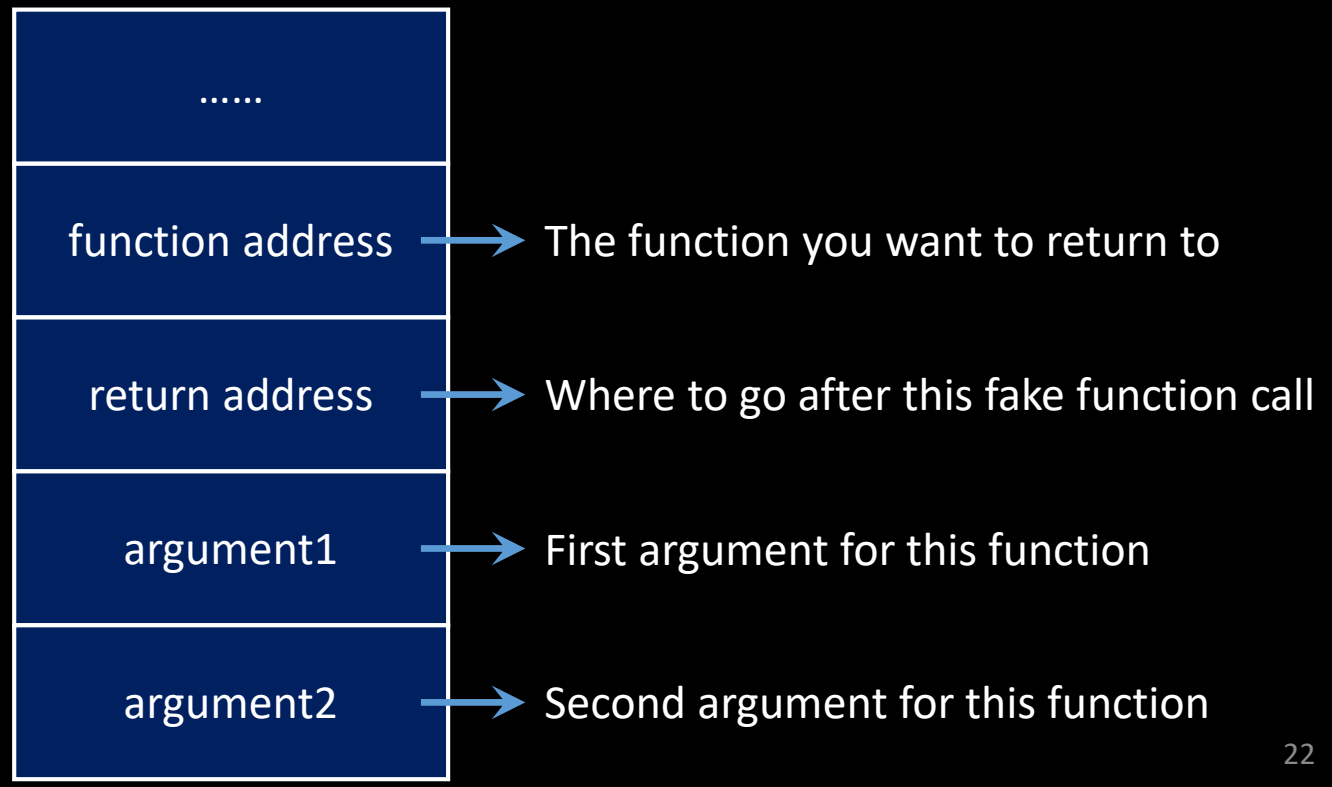
```
.....
mov     DWORD PTR [esp+0x4],0x804a020
mov     DWORD PTR [esp],0x804a035
call    80482f0 <strcpy@plt>
.....
```

Source address: 0x804a020
Destination address: 0x804a035



ret2libc

- When calling a function, there will be some needed information on stack.
- If we know a function's address, we can fake a function call by return to the function with proper value on stack.

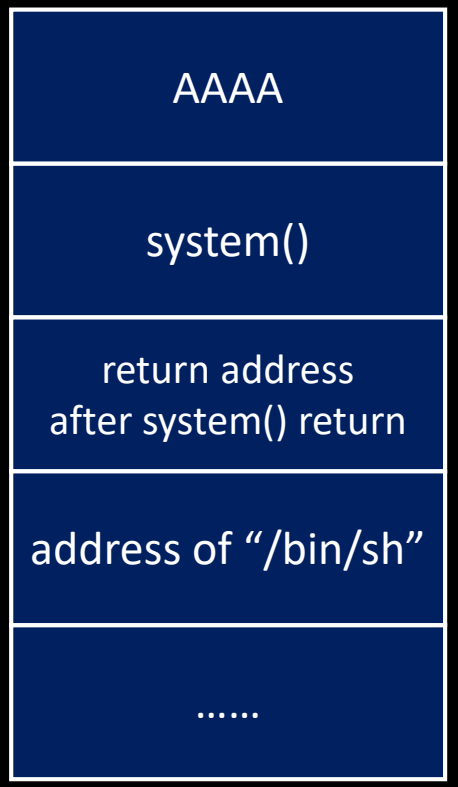


fake a function call

- Put needed value on stack when buffer overflow .
- After returning to the function, it can fetch right data from right position, just like a normal function call !

fake a function call

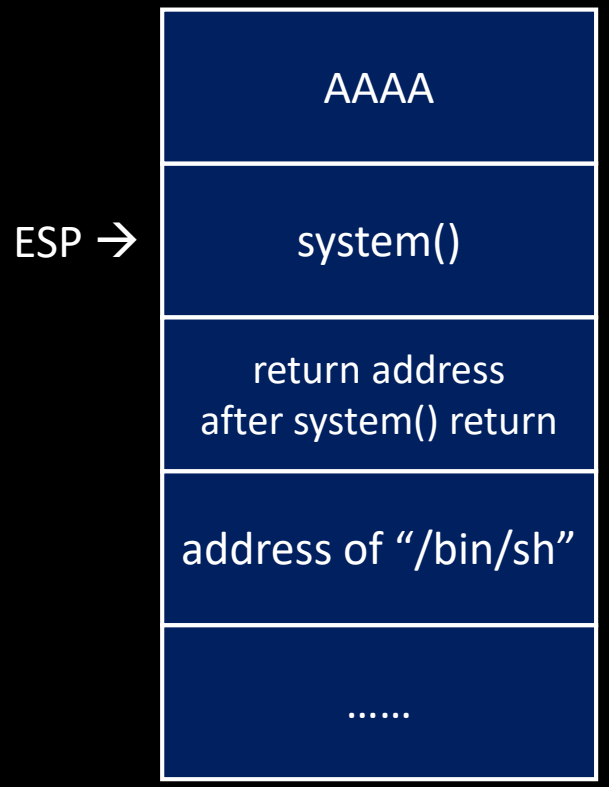
- Put needed value on stack when buffer overflow .
- After returning to the function, it can fetch right data from right position, just like a normal function call !



fake a function call

- Put needed value on stack when buffer overflow .
- After returning to the function, it can fetch right data from right position, just like a normal function call !

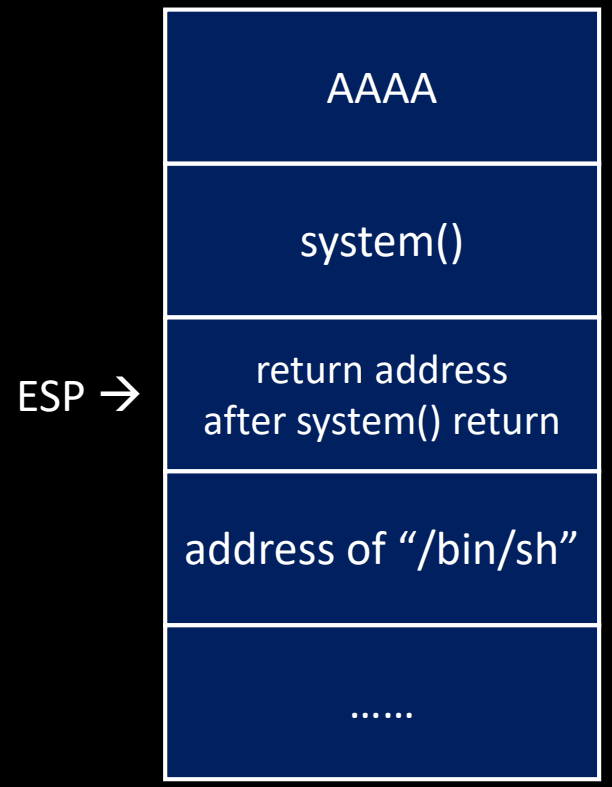
```
……
leave
ret
……
```



fake a function call

- Put needed value on stack when buffer overflow .
- After returning to the function, it can fetch right data from right position, just like a normal function call !

```
……
leave
ret
……
```



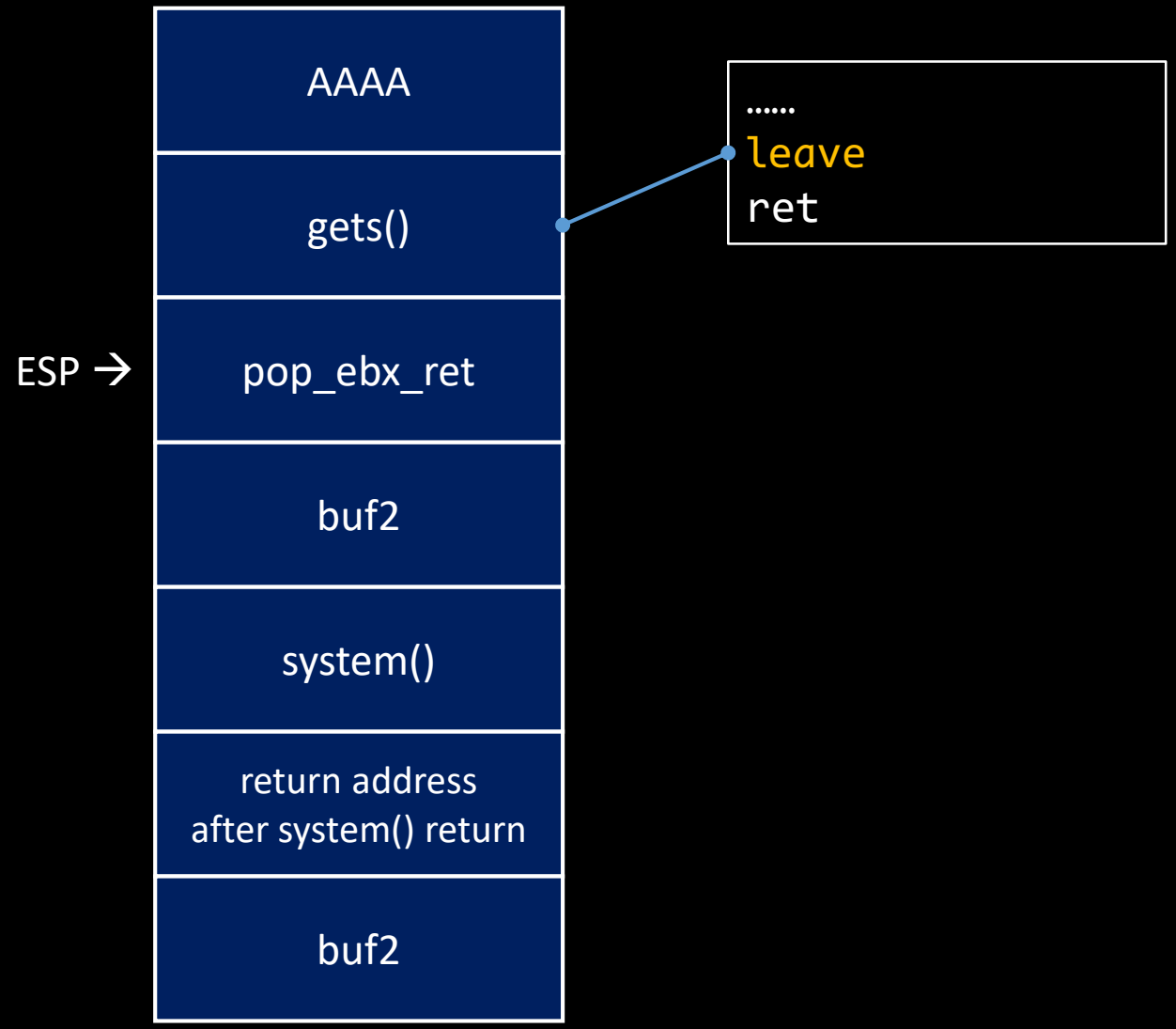
ret2libc

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

char *shell = "/bin/sh";
char buf2[100];

void secure(void)
{
    int secretcode, input;
    srand(time(NULL));

    secretcode = rand();
    scanf("%d", &input);
    if(input == secretcode)
        system("shell!?");
}

int main(void)
{
    setvbuf(stdout, 0LL, 2, 0LL);
    setvbuf(stdin, 0LL, 1, 0LL);

    char buf1[100];

    printf("RET2LIBC >_<\n");
    gets(buf1);

    return 0;
}
```

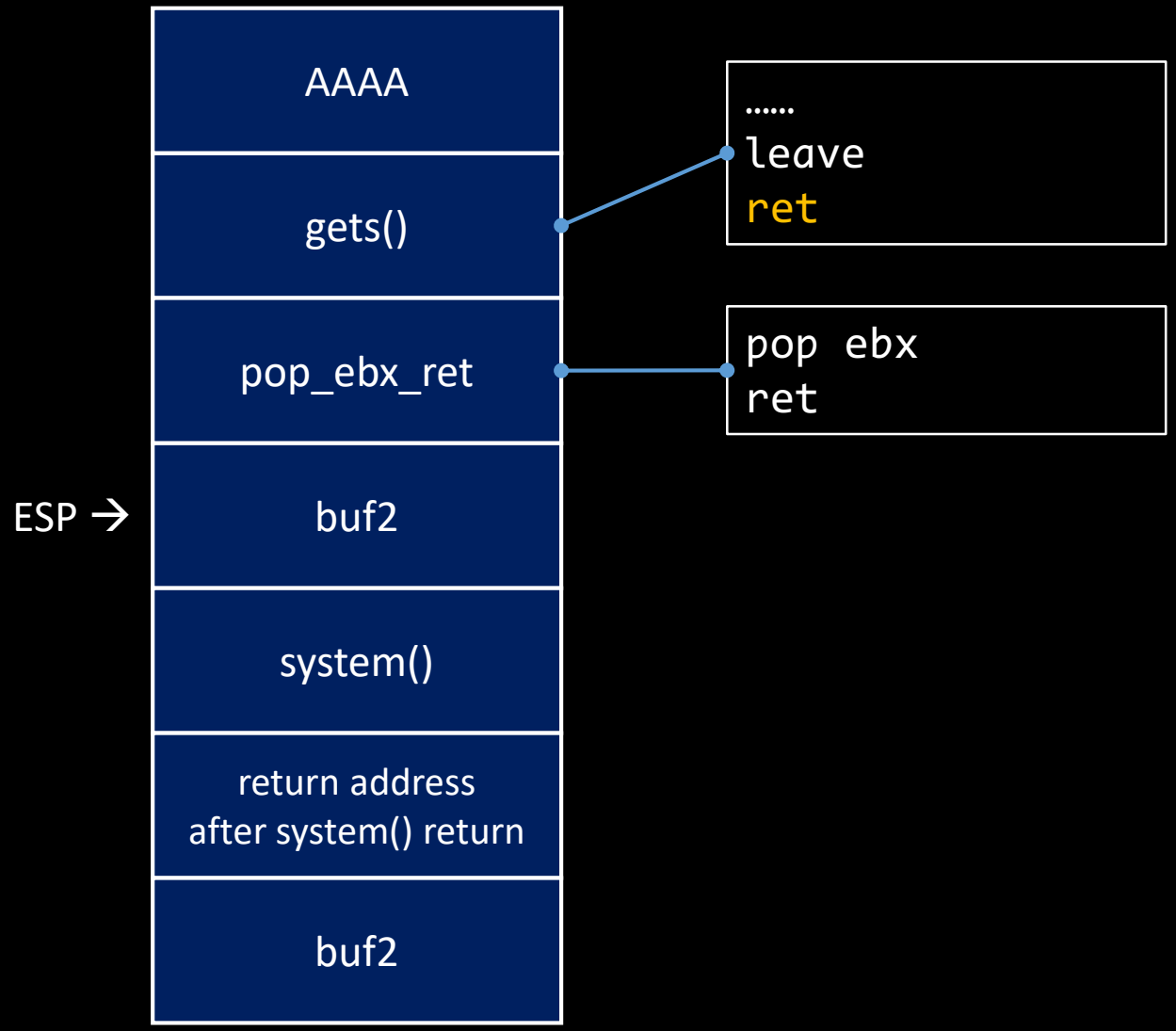
ret2libc

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

char buf2[100];

void secure(void)
{
    int secretcode, input;
    srand(time(NULL));

    secretcode = rand();
    scanf("%d", &input);
    if(input == secretcode)
        system("no_shell_QQ");
}

int main(void)
{
    setvbuf(stdout, 0LL, 2, 0LL);
    setvbuf(stdin, 0LL, 1, 0LL);

    char buf1[100];

    printf("Something surprise here, but I don't think it will work.\n");
    printf("What do you think ?");
    gets(buf1);

    return 0;
}
```

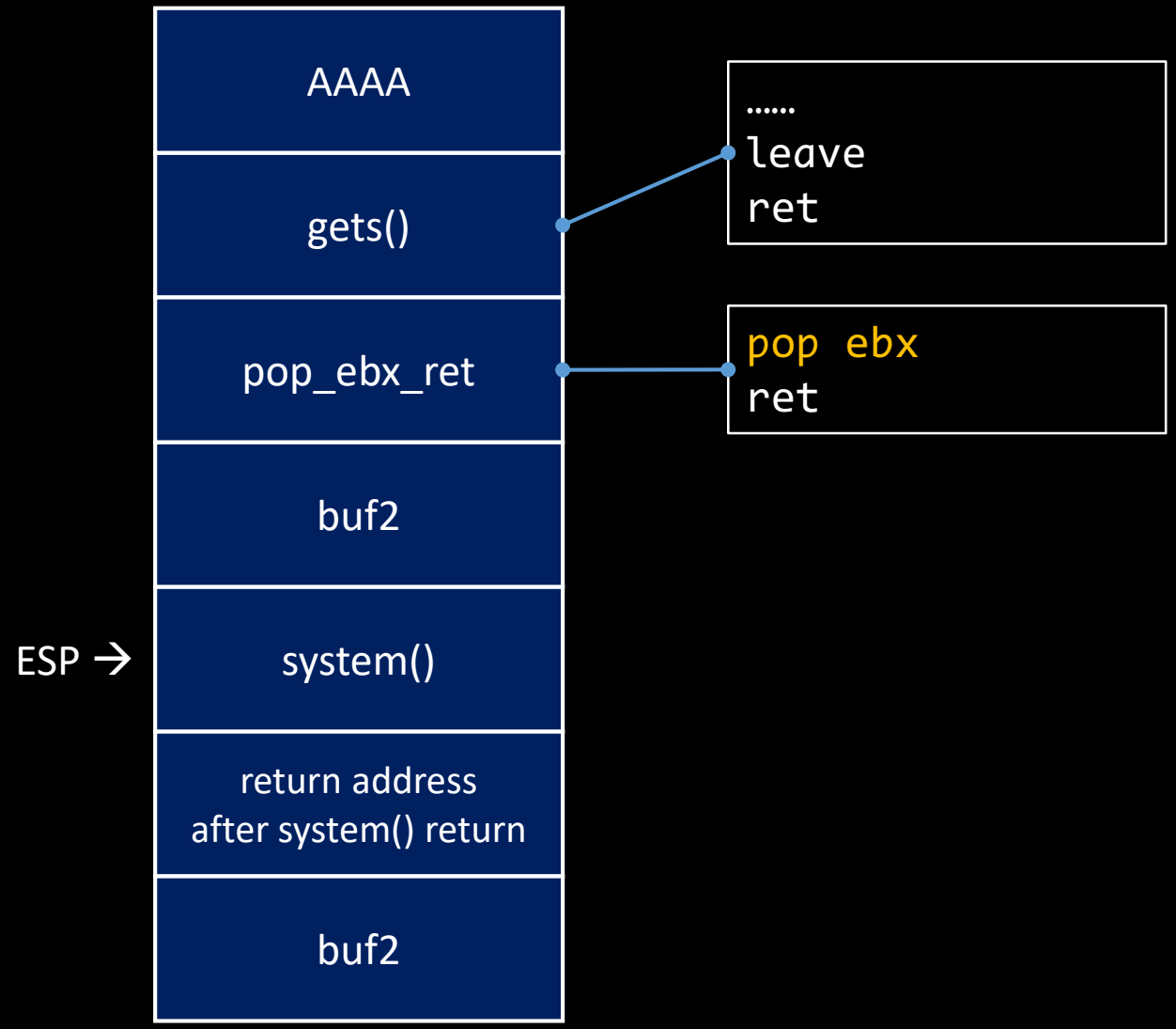
ret2libc


AAAA
gets()
ESP →
pop_ebx_ret
buf2
system()
return address
after system() return
buf2
.....
leave
ret

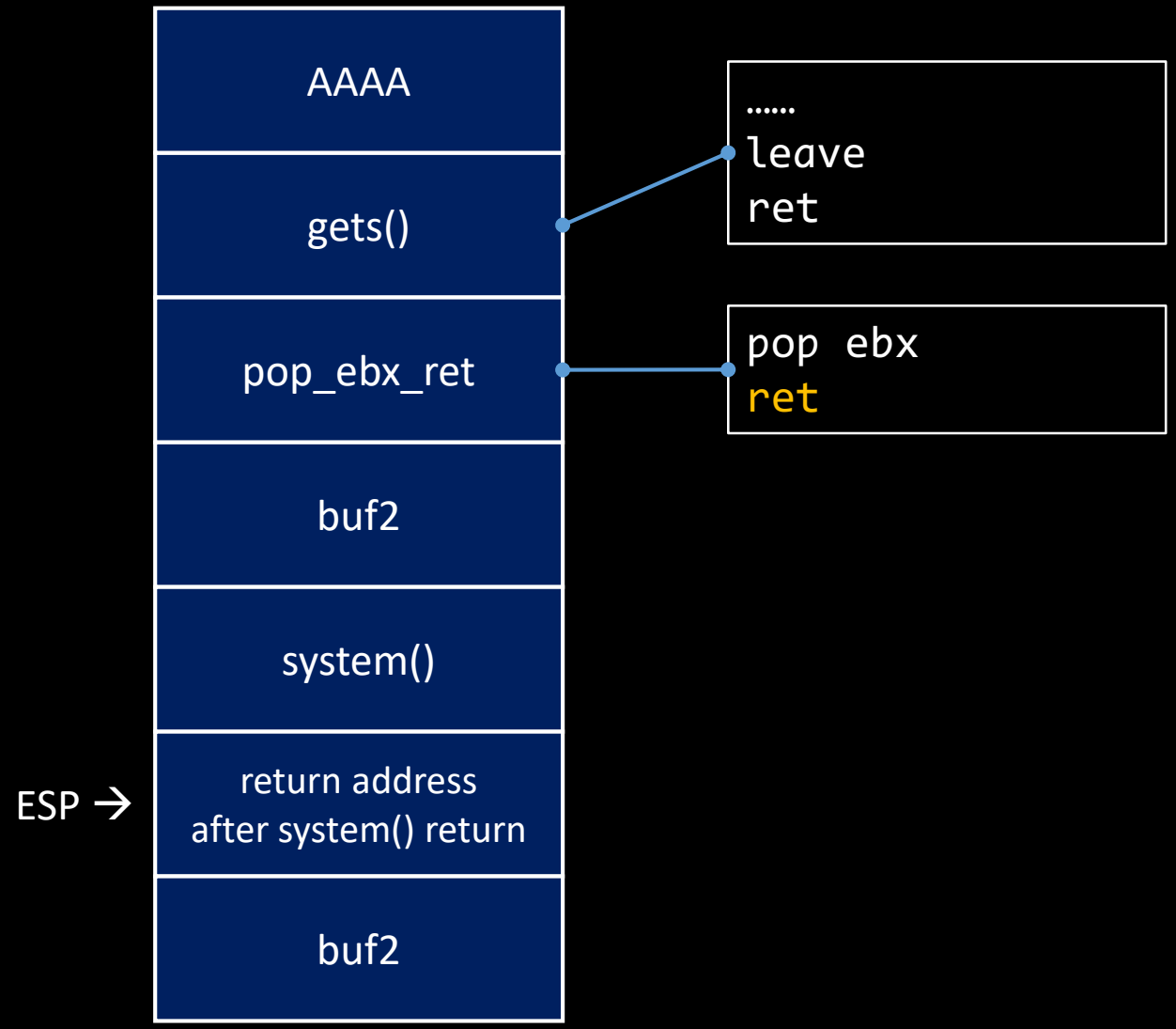
ret2libc


AAAA
gets()
pop_ebx_ret
buf2
system()
return address
after system() return
buf2
ESP →
……
leave
ret
pop ebx
ret

ret2libc


AAAA
gets()
pop_ebx_ret
buf2
ESP →
system()
return address
after system() return
buf2
……
leave
ret
pop ebx
ret

ret2libc


AAAA
gets()
pop_ebx_ret
buf2
system()
return address
after system() return
buf2
ESP →
……
leave
ret
pop ebx
ret

ASLR and libc address

- With ASLR, the base address of libc is randomized each time we execute a program .
- We can’t predict the function’s address we want to use and return to it, except those used by this program .

```
08048000-08049000 r-xp 00000000 08:01 928487                             /root/CSC/ROP/ret2text
08049000-0804a000 r--p 00000000 08:01 928487                             /root/CSC/ROP/ret2text
0804a000-0804b000 rw-p 00001000 08:01 928487                             /root/CSC/ROP/ret2text
08dc2000-08de3000 rw-p 00000000 00:00 0                                  [heap]
f75a6000-f75a7000 rw-p 00000000 00:00 0
f75a7000-f7753000 r-xp 00000000 08:01 3014705                            /lib32/libc-2.23.so
f7753000-f7754000 ---p 001ac000 08:01 3014705                            /lib32/libc-2.23.so
f7754000-f7756000 r--p 001ac000 08:01 3014705                            /lib32/libc-2.23.so
f7756000-f7757000 rw-p 001ae000 08:01 3014705                            /lib32/libc-2.23.so
f7757000-f775b000 rw-p 00000000 00:00 0
f777a000-f777c000 r--p 00000000 00:00 0                                  [vvar]
f777c000-f777d000 r-xp 00000000 00:00 0                                  [vdso]
f777d000-f779f000 r-xp 00000000 08:01 3014701                            /lib32/ld-2.23.so
f779f000-f77a0000 rw-p 00000000 00:00 0
f77a0000-f77a1000 r--p 00022000 08:01 3014701                            /lib32/ld-2.23.so
f77a1000-f77a2000 rw-p 00023000 08:01 3014701                            /lib32/ld-2.23.so
ffa90000-ffab1000 rw-p 00000000 00:00 0                                  [stack]
```

How to find functions' offset

- The offset of a function in a libc will be fixed .
- Function's address will be **libc base + function's offset** .
- readelf or pwntools can help you find the offset !

```
192: 001288b0    91 FUNC    GLOBAL DEFAULT   12 getnetbyname_r@GLIBC_2.0
193: 0007a400    82 IFUNC   GLOBAL DEFAULT   12 strncmp@@GLIBC_2.0
194: 000fd0e0   607 FUNC    GLOBAL DEFAULT   12 getnetbyname_r@@GLIBC_2.1.2
195: 000fd860   211 FUNC    GLOBAL DEFAULT   12 getprotoent_r@@GLIBC_2.1.2
196: 001175a0    17 FUNC    GLOBAL DEFAULT   12 svcfd_create@@GLIBC_2.0
197: 000e3d20    58 FUNC    WEAK   DEFAULT   12 ftruncate@@GLIBC_2.0
198: 00128970    53 FUNC    GLOBAL DEFAULT   12 getprotoent_r@GLIBC_2.0
199: 00081150    44 FUNC    GLOBAL DEFAULT   12 __strncpy_gg@@GLIBC_2.1.1
200: 0010f4c0   138 FUNC    GLOBAL DEFAULT   12 xdr_unixcred@@GLIBC_2.1
201: 00029730    72 FUNC    WEAK   DEFAULT   12 dcngettext@@GLIBC_2.2
202: 0010cd50   125 FUNC    GLOBAL DEFAULT   12 xdr_rmtcallres@@GLIBC_2.0
203: 00064da0   421 FUNC    GLOBAL DEFAULT   12 _IO_puts@@GLIBC_2.0
204: 001075d0   242 FUNC    GLOBAL DEFAULT   12 inet_nsap_addr@@GLIBC_2.0
205: 00106de0   286 FUNC    WEAK   DEFAULT   12 inet_aton@@GLIBC_2.0
206: 000e4890   217 FUNC    GLOBAL DEFAULT   12 ttyslot@@GLIBC_2.0
207: 001aa8dc     4 OBJECT  GLOBAL DEFAULT   32 __rcmd_errstr@@GLIBC_2.0
208: 000d7dd0    90 FUNC    GLOBAL DEFAULT   12 wordfree@@GLIBC_2.1
```

`readelf -s file_name`

```
>>> from pwn import *
>>> libc = ELF('libc.so.6')
[*] '/root/CSC/ROP/libc.so.6'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
>>> print hex(libc.symbols['strncmp'])
0x7a400
>>>
```

How to find libc's base address

- For those dynamically linked program, when a function is called once, its address in libc will be written to its GOT entry .

`file` *`file_name`*

```
ret2text: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 2.6.24, BuildID[
```

`objdump -d` *`file_name`*

```
080482f0 <strcpy@plt>:
 80482f0:       ff 25 0c a0 04 08       jmp    DWORD PTR ds:0x804a00c
 80482f6:       68 00 00 00 00          push   0x0
 80482fb:       e9 e0 ff ff ff          jmp    80482e0 <_init+0x2c>

08048300 <__gmon_start__@plt>:
 8048300:       ff 25 10 a0 04 08       jmp    DWORD PTR ds:0x804a010
 8048306:       68 08 00 00 00          push   0x8
 804830b:       e9 d0 ff ff ff          jmp    80482e0 <_init+0x2c>

08048310 <__libc_start_main@plt>:
 8048310:       ff 25 14 a0 04 08       jmp    DWORD PTR ds:0x804a014
 8048316:       68 10 00 00 00          push   0x10
 804831b:       e9 c0 ff ff ff          jmp    80482e0 <_init+0x2c>
```

<strcpy@got.plt> , this entry will store :

next instruction in <strcpy@plt>
if strcpy haven't been called yet .

strcpy()'s address in libc !
if strcpy have been called .

How to find libc's base address

- For those dynamically linked program, when a function is called once, its address in libc will be written to its GOT entry .

`file` *`file_name`*

```
ret2text: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 2.6.24, BuildID[
```

`objdump -d` *`file_name`*

```
080482f0 <strcpy@plt>:
 80482f0:       ff 25 0c a0 04 08       jmp    DWORD PTR ds:0x804a00c
 80482f6:       68 00 00 00 00          push   0x0
 80482fb:       e9 e0 ff ff ff          jmp    80482e0 <_init+0x2c>

08048300 <__gmon_start__@plt>:
 8048300:       ff 25 10 a0 04 08       jmp    DWORD PTR ds:0x804a010
 8048306:       68 08 00 00 00          push   0x8
 804830b:       e9 d0 ff ff ff          jmp    80482e0 <_init+0x2c>

08048310 <__libc_start_main@plt>:
 8048310:       ff 25 14 a0 04 08       jmp    DWORD PTR ds:0x804a014
 8048316:       68 10 00 00 00          push   0x10
 804831b:       e9 c0 ff ff ff          jmp    80482e0 <_init+0x2c>
```

<strcpy@got.plt> , this entry will store :

next instruction in <strcpy@plt>
if strcpy haven't been called yet .

strcpy()'s address in libc !
if strcpy have been called .

How to find libc’s base address

- We can use `puts` to leak a function’s address store in GOT entry .
- Subtract this function’s offset in libc from the leaked address, we can get libc’s base address !
- Then we can know each function’s address :
  libc base + function’s offsets

ret2libc

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

char buf2[100];

void secure(void)
{
    int secretcode, input;
    srand(time(NULL));

    secretcode = rand();
    scanf("%d", &input);
    if(input == secretcode)
        puts("no_shell_QQ");
}

int main(void)
{
    setvbuf(stdout, 0LL, 2, 0LL);
    setvbuf(stdin, 0LL, 1, 0LL);

    char buf1[100];

    printf("No surprise anymore, system disappeard QQ.\n");
    printf("Can you find it !?");
    gets(buf1);

    return 0;
}
```

What is the next step ?

- ret2text
- Return to shellcode
- ret2libc
- ROP

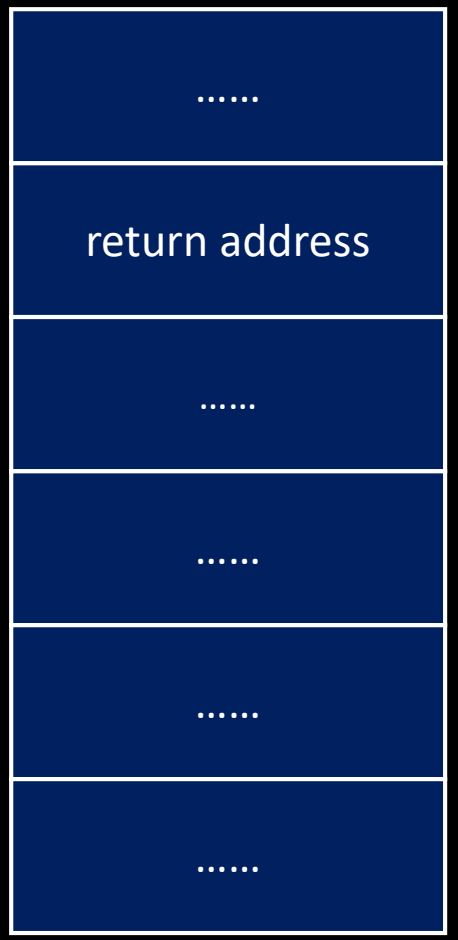
ROP

- Try to use gadgets, some machine instruction sequences which end with `ret`, to achieve our goal .

```
8048729:       83 c4 1c                add    esp,0x1c
804872c:       5b                      pop    ebx
804872d:       5e                      pop    esi
804872e:       5f                      pop    edi
804872f:       5d                      pop    ebp
8048730:       c3                      ret
```

For example, this is a gadget comes from <__libc_csu_init>

ROP

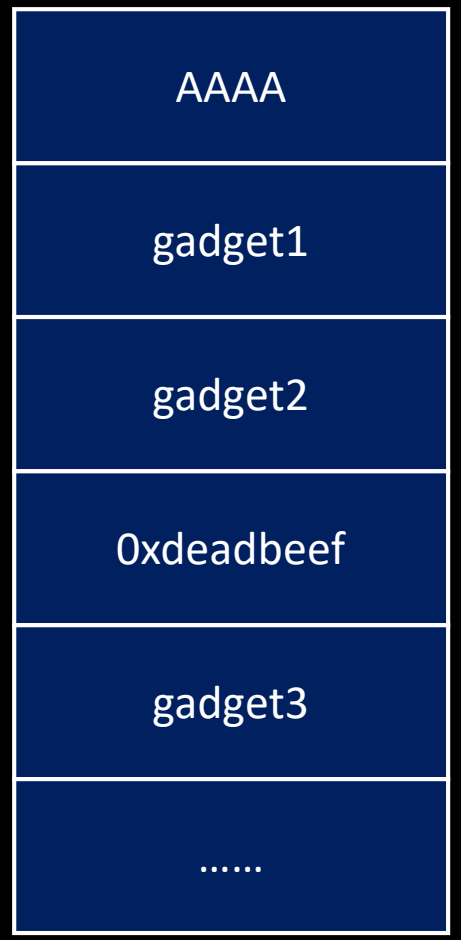


ROP

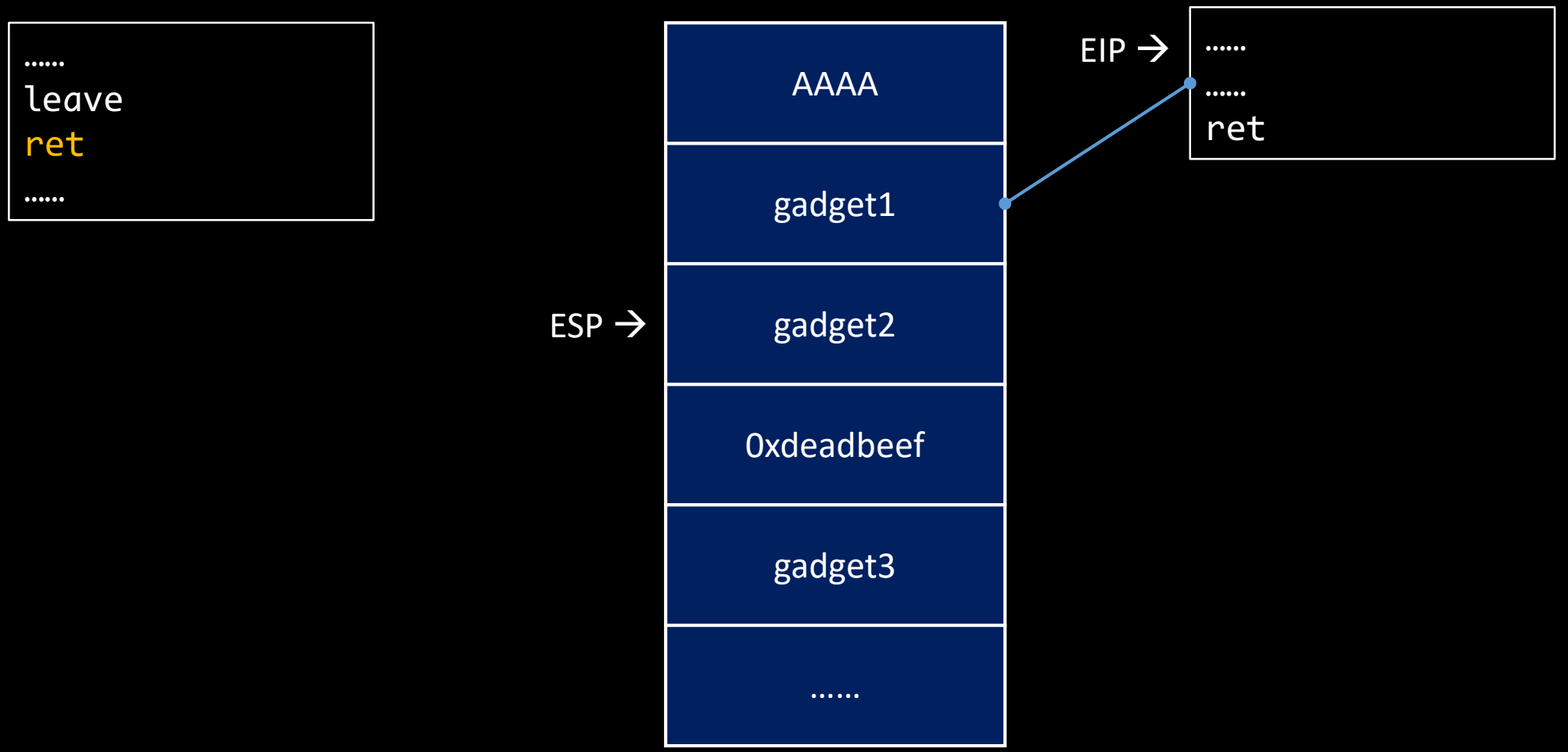
ROP

```
……
leave
ret
……
```

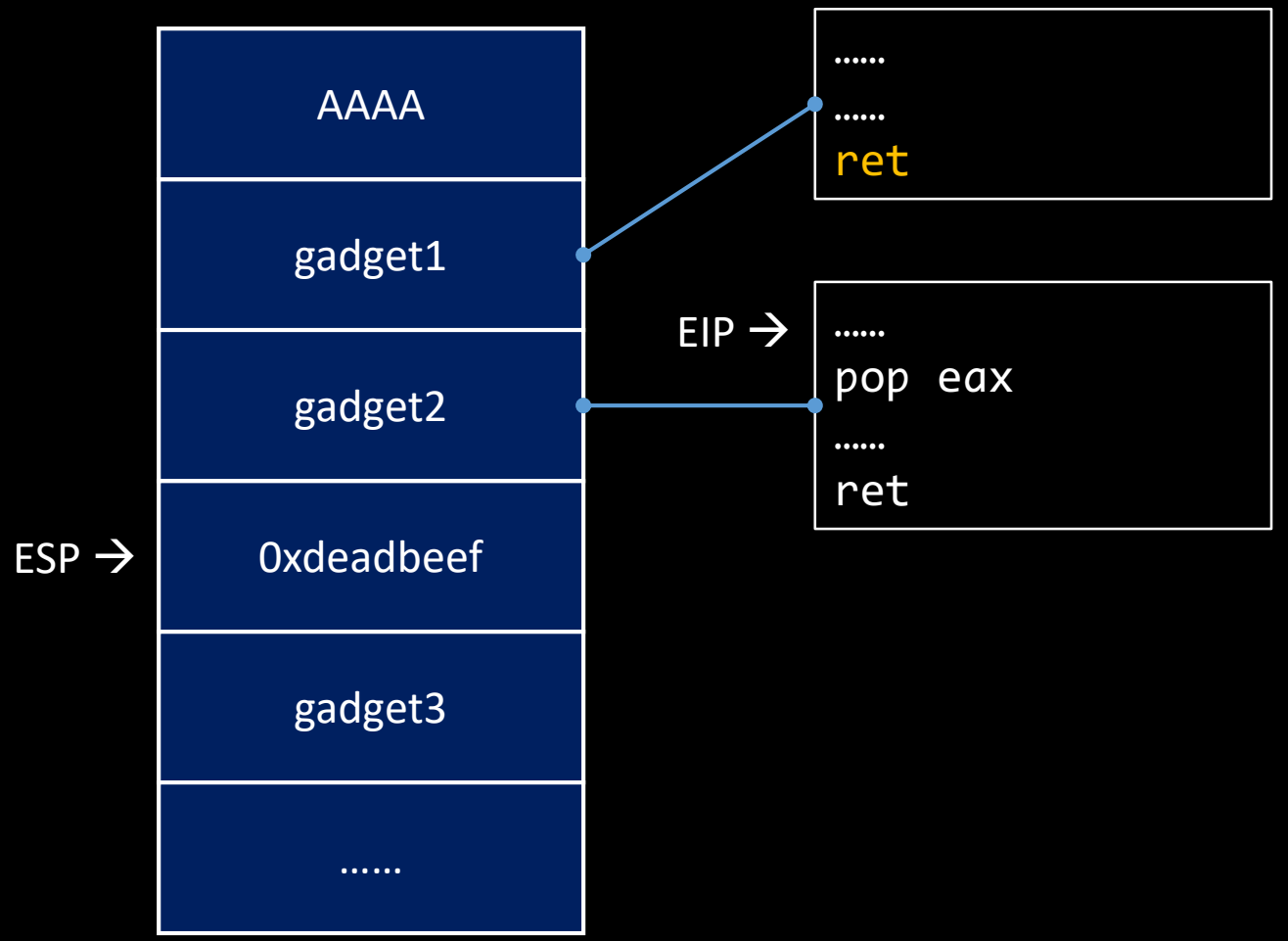
ESP →



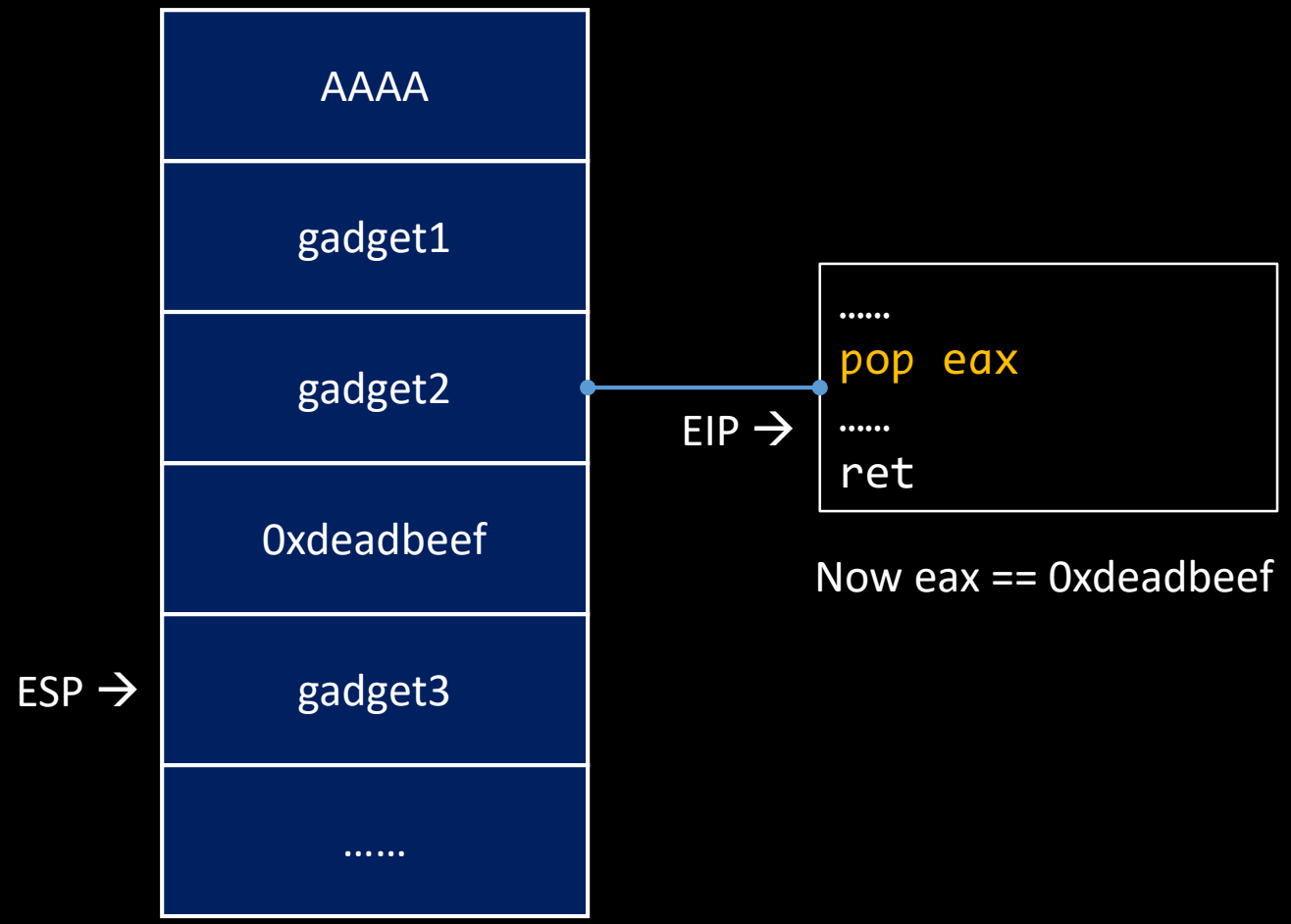
ROP


……
leave
ret
……
ESP →
AAAA
gadget1
gadget2
0xdeadbeef
gadget3
……
EIP →
……
……
ret

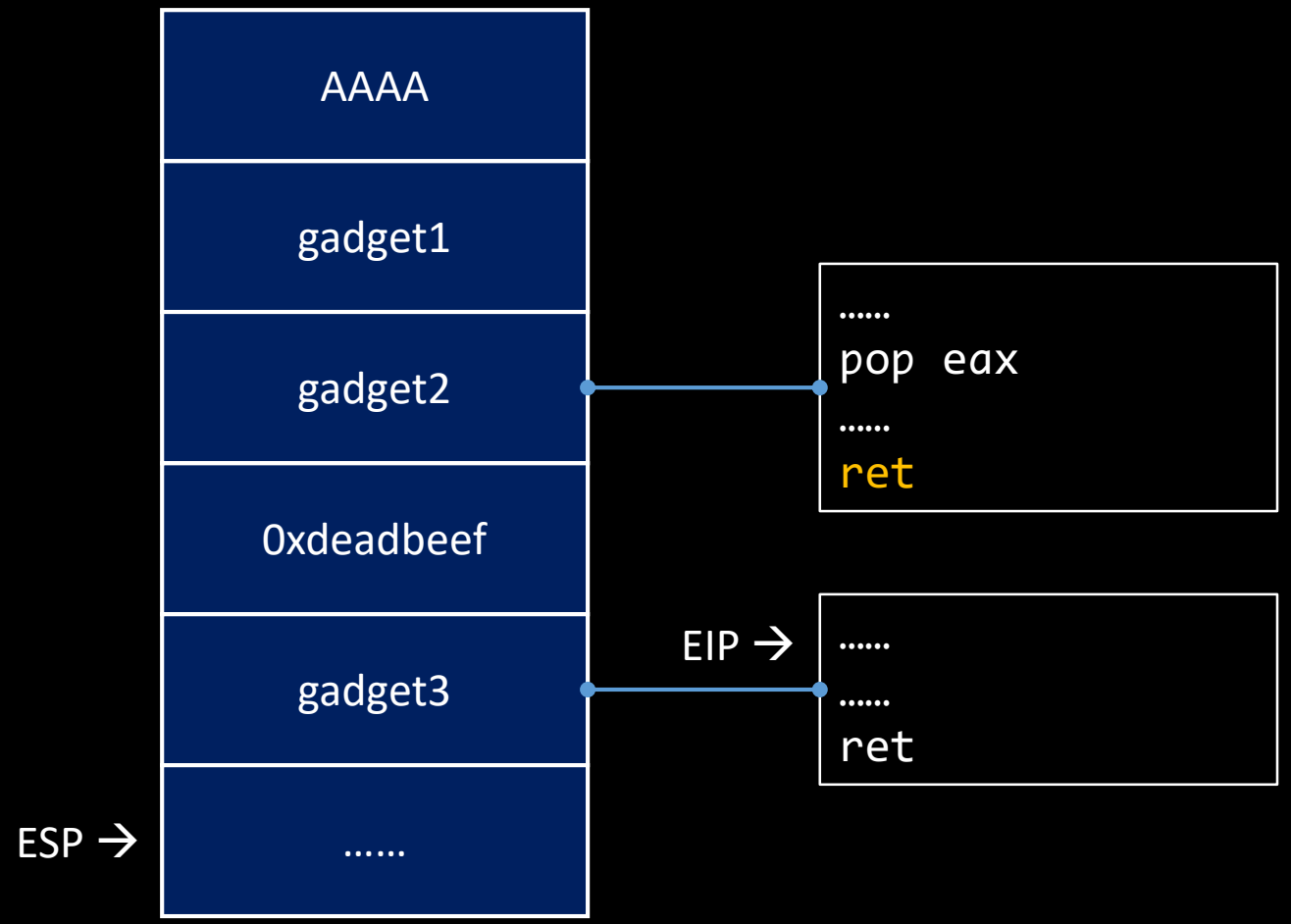
ROP


AAAA
gadget1
gadget2
0xdeadbeef
gadget3
……
ESP →
EIP →
……
……
ret
……
pop eax
……
ret

ROP


AAAA
gadget1
gadget2
0xdeadbeef
gadget3
……
ESP →
EIP →
……
pop eax
……
ret
Now eax == 0xdeadbeef

ROP


AAAA
gadget1
gadget2
0xdeadbeef
gadget3
……
ESP →
……
pop eax
……
ret
EIP →
……
……
ret

Find available gadgets

- ROPgadget can help you find gadgets .
- Usage:
  ROPgadget --binary *file_name*
  ROPgadget --binary *file_name* --opcode *opcode*

```
Gadgets information
============================================================
0x080488a3 : adc al, 0x41 ; ret
0x08048560 : add al, 0x24 ; inc eax ; mov al, byte ptr [0xd0ff0804] ; leave ; ret
0x0804859d : add al, 0x24 ; inc eax ; mov al, byte ptr [0xd2ff0804] ; leave ; ret
0x080485c8 : add al, 8 ; add ecx, ecx ; ret
0x08048564 : add al, 8 ; call eax
0x080485a1 : add al, 8 ; call edx
0x08048548 : add al, 8 ; cmp eax, 6 ; ja 0x8048557 ; ret
0x080486c0 : add byte ptr [eax], al ; add byte ptr [eax], al ; leave ; ret
0x080486c1 : add byte ptr [eax], al ; add cl, cl ; ret
0x08048438 : add byte ptr [eax], al ; add esp, 8 ; pop ebx ; ret
0x080486c2 : add byte ptr [eax], al ; leave ; ret
0x080488a0 : add cl, byte ptr [eax + 0xe] ; adc al, 0x41 ; ret
0x080486c3 : add cl, cl ; ret
0x0804889c : add eax, 0x2300e4e ; dec eax ; push cs ; adc al, 0x41 ; ret
0x080485c5 : add eax, 0x804a064 ; add ecx, ecx ; ret
0x08048582 : add eax, edx ; sar eax, 1 ; jne 0x804858f ; ret
0x080485ca : add ecx, ecx ; ret
```

ROP

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

char *shell = "/bin/sh";

int main(void)
{
    setvbuf(stdout, 0LL, 2, 0LL);
    setvbuf(stdin, 0LL, 1, 0LL);

    char buf[100];

    printf("This time, no system() and NO SHELLCODE!!!\n");
    printf("What do you plan to do?\n");
    gets(buf);

    return 0;
}
```

Reference

- ROP
- Mechanisms preventing buffer overflow
- pwntools packing and unpacking of strings
- ROPgadget
- Linux syscall reference

Practice

- train.cs.nctu.edu.tw
  - ret2libc
  - ROP
- secprog.cs.nctu.edu.tw
  - 10/21 practice BOF1
  - 10/21 practice BOF2
  - 10/21 practice BOF3
  - 11/4 practice BOF4
  - 11/4 practice BOF6