lab1：misc编解码及流量取证

1. Task 1 (25%)

`calculator.lua` 是一个计算器程序，但文件末尾有大量空行，最后一行写了`-- nope no flag here, keep looking :P`。

Image 1

flag 可能被隐藏在空白行中。选择文字，可以发现这些"空白行"是由**空格（Space, 0x20）**和**制表符（Tab, 0x09）**组成的二进制编码数据。

使用 `cat -et` 显示，空格为空格，Tab 为 `^I`：

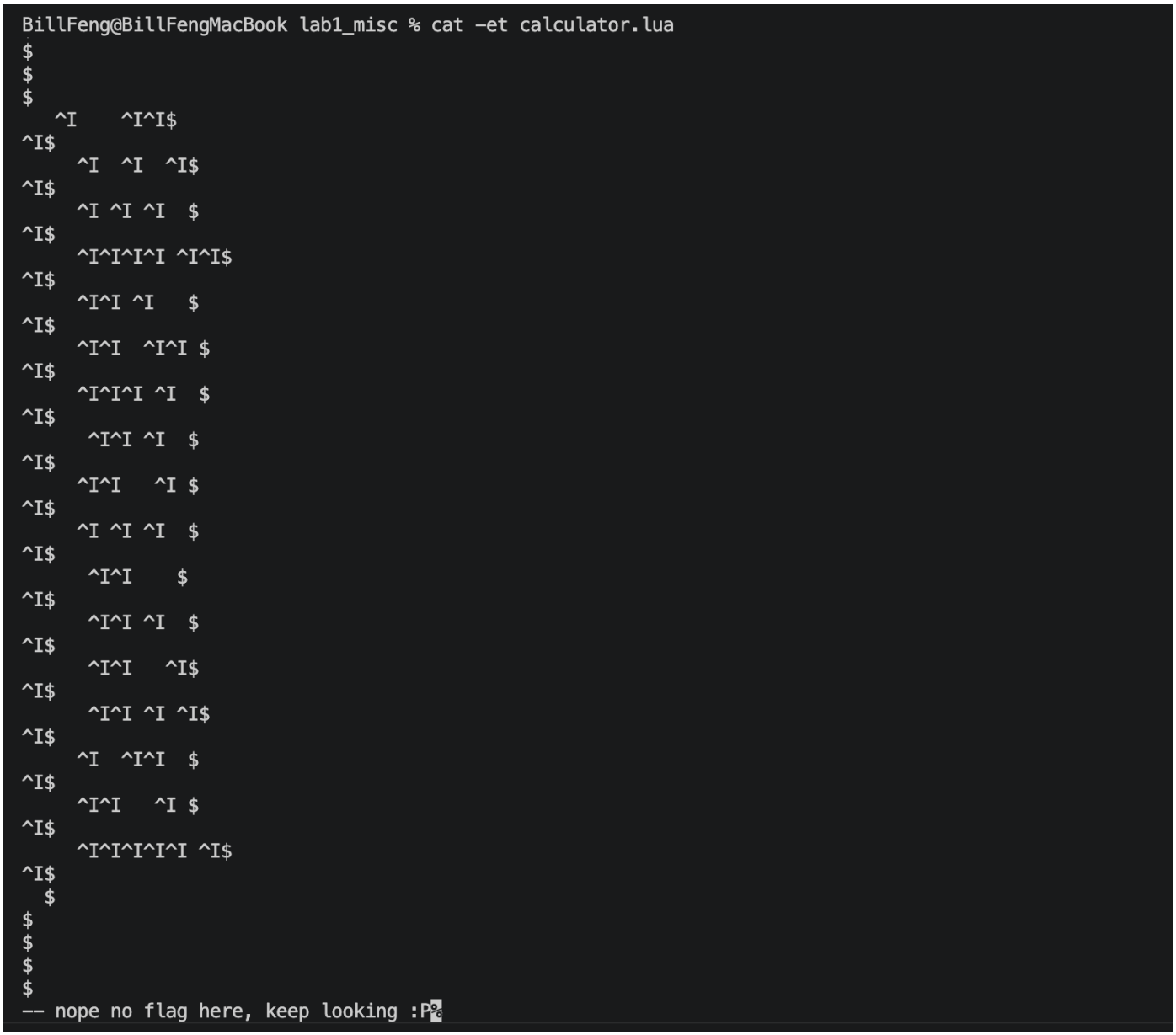

Image 2

1.1 猜测编码格式并解码

- Space（空格，`0x20`） → bit 0
- Tab（制表符，`0x09`） → bit 1

每隔一行是一个数据行，数据行之间由仅含单个 Tab 的分隔行隔开。每个数据行包含 10–12 个空白字符，前导若干个空格（bit 0），取最后连续 7 位即为一个 ASCII 字符的二进制表示。

行号	完整二进制序列	后 7 位（有效数据）	ASCII	字符
80	0001000011	1000011	67	C
82	000001001001	1001001	73	I
84	000001010100	1010100	84	T
88	000001101000	1101000	104	h
…	…	…	…	…
106	000000110101	0110101	53	5
108	000001001100	1001100	76	L
110	000001100010	1100010	98	b
112	000001111101	1111101	125	}

Table 1

`CIT{hft4bT0415Lb}`

1.2 python脚本

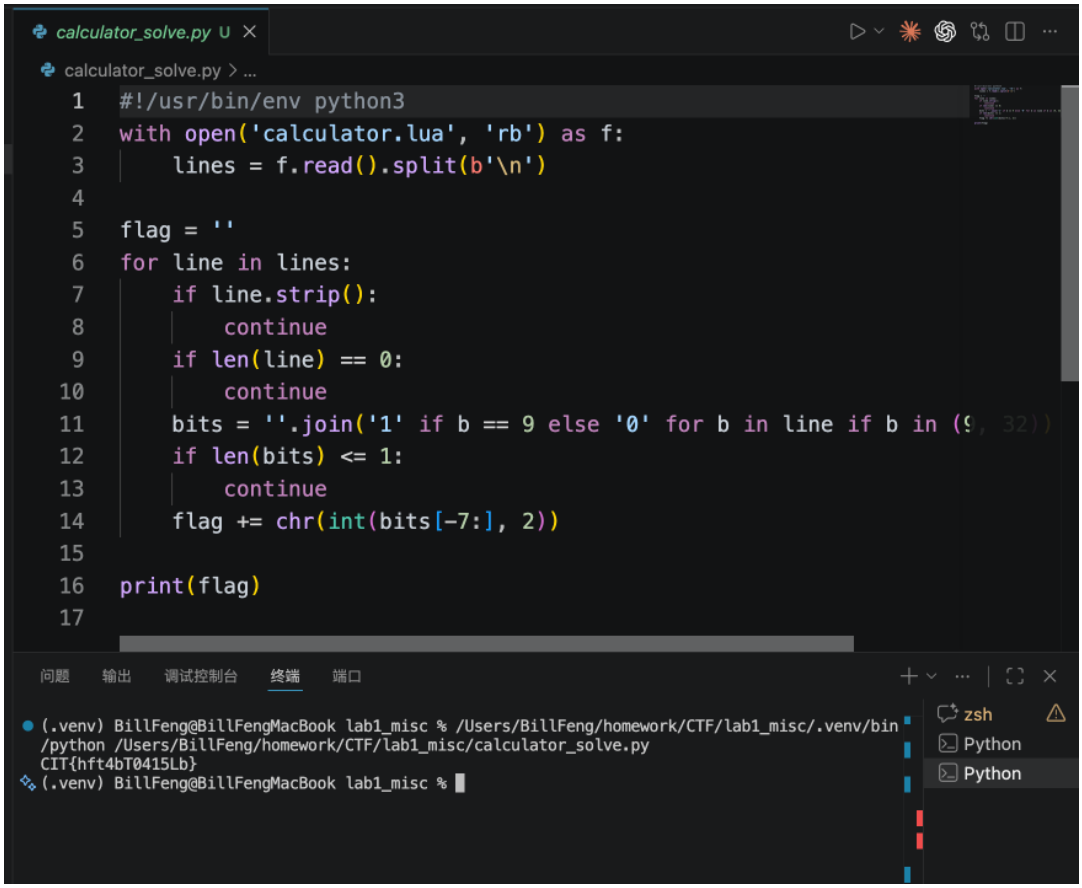

Image 3

1. `for b in line` 遍历一行的每个字节
2. `if b in (9, 32)` 只保留 Tab（9）和空格（32）
3. `'1' if b == 9 else '0'` Tab `'1'`，空格 `'0'`

得到 flag：`CIT{hft4bT0415Lb}`

2. Task 2 (25%)

关于Base64：

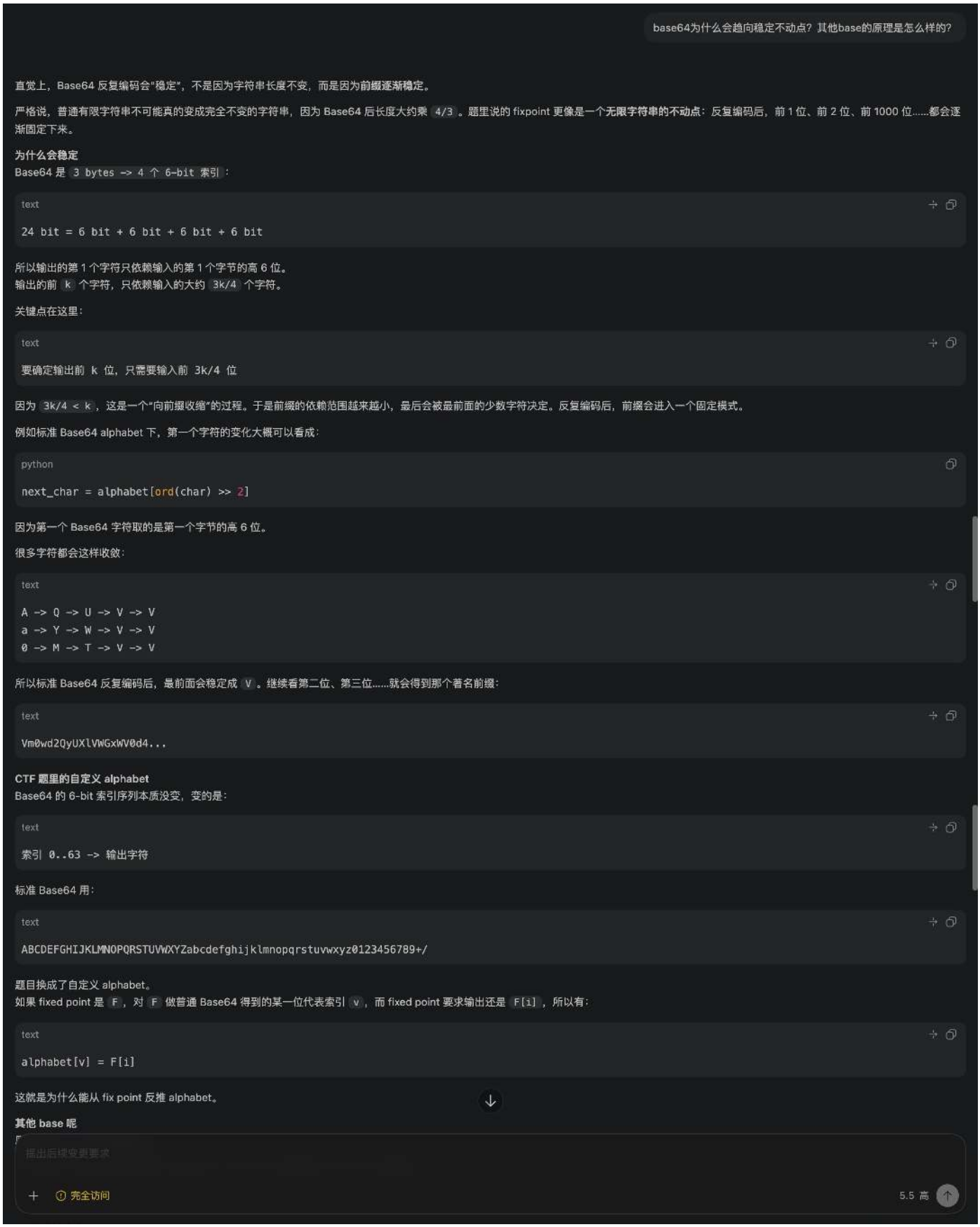

Image 4

题目给出自定义 Base64 alphabet（同时也是flag）：

`bctf{??????????????????????FiXed??????????????????????p01nT}`

并给出在该 alphabet 下反复 Base64 编码后得到的 fix point 前缀。

2.1 标准Base64

Base64 每次处理 3 个字节，即 24 bit。把这 24 bit 从高位到低位切成 4 组，每组 6 bit：

- 第 1 组：第 23 到 18 位
- 第 2 组：第 17 到 12 位
- 第 3 组：第 11 到 6 位
- 第 4 组：第 5 到 0 位

每一组 6 bit 的取值范围是 `0..63`，这个值就是 alphabet 的索引。标准 Base64 会用 `ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/` 中对应位置的字符输出；本题则把同一个索引映射到自定义 alphabet 中对应位置的字符。

例如 3 字节 `abc` 的二进制为：

`01100001 01100010 01100011`

切成 4 组 6 bit：

`011000 010110 001001 100011`

对应十进制索引：

`24, 22, 9, 35`

标准 alphabet 下得到 `YWJj`，因为标准 alphabet 的第 `24, 22, 9, 35` 个字符分别是 `Y, W, J, j`。

2.2 建立映射

题目编码函数：

1. 先对当前字符串做普通 Base64 编码，得到一串标准 Base64 字符。
2. 去掉 `=` padding。

> 什么是 padding?
>
> ```
> "abc" -> YWJj     # 3 字节，不加 =
> "ab"  -> YWI=     # 2 字节，加 1 个 =
> "a"   -> YQ==     # 1 字节，加 2 个 =
> ```

3. 再把标准 alphabet 的每个字符按相同索引翻译成自定义 alphabet 的字符。

设题目给出的 fixed point 为 `F`，自定义 alphabet 为 `A`。因为它是 fixed point，所以：

`custom_base64(F) = F`

也就是说，对 `F` 做普通 Base64 时，第 `i` 个输出字符代表的 6-bit 索引如果是 `v`，那么自定义编码后的第 `i` 个字符就是 `A[v]`。而 fixed point 要求这个字符等于原来的 `F[i]`，因此可以得到：

```
A[v] = F[i]
```

所以将 fixed point 做一次标准 base64，然后翻译成索引，又已知在自定义 alphabet 下的 base64（正是它本身，fixed），重新建立映射即可。

2.3 python脚本

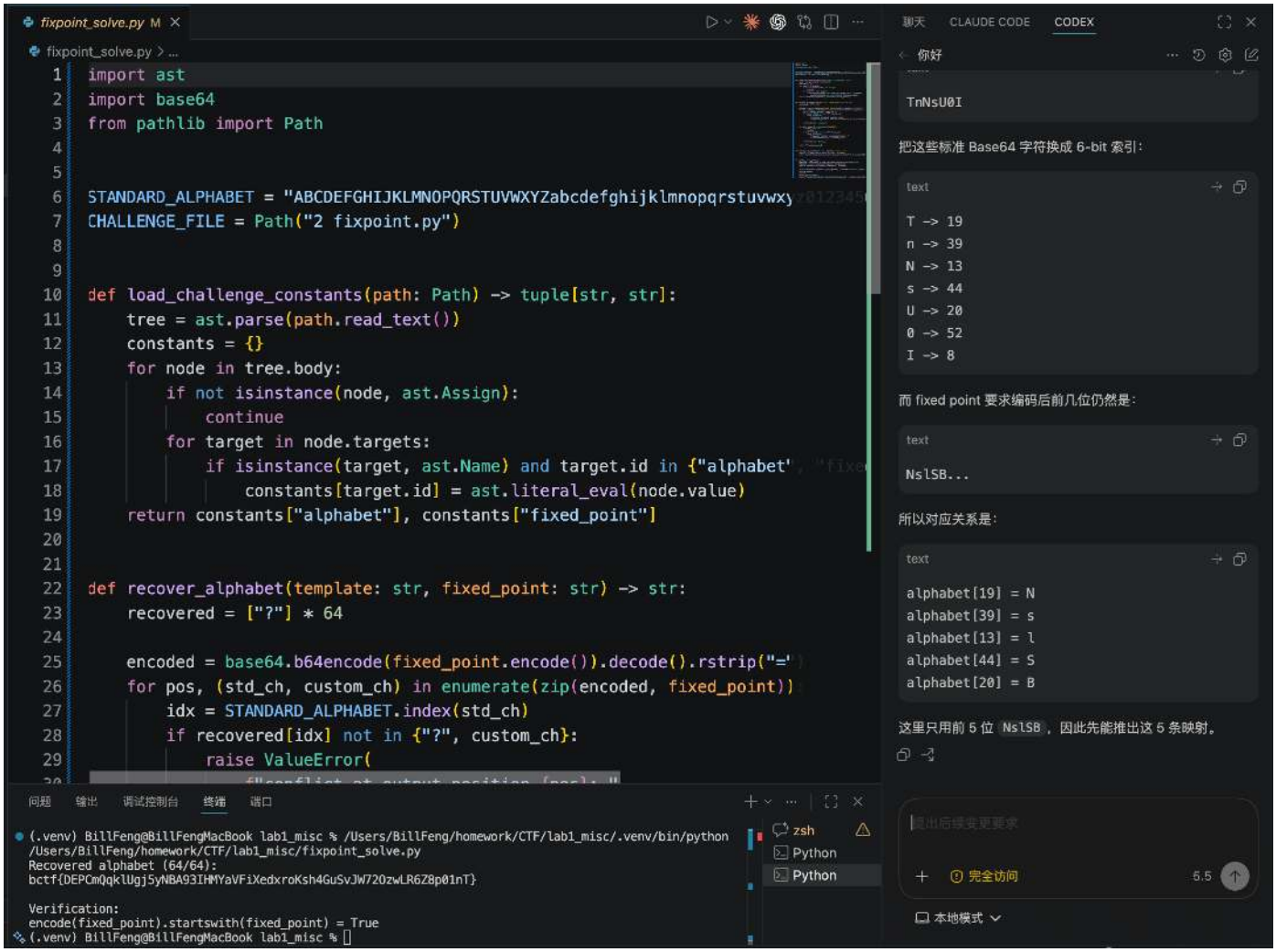

Image 5

通过脚本实现上述过程。最后验证。

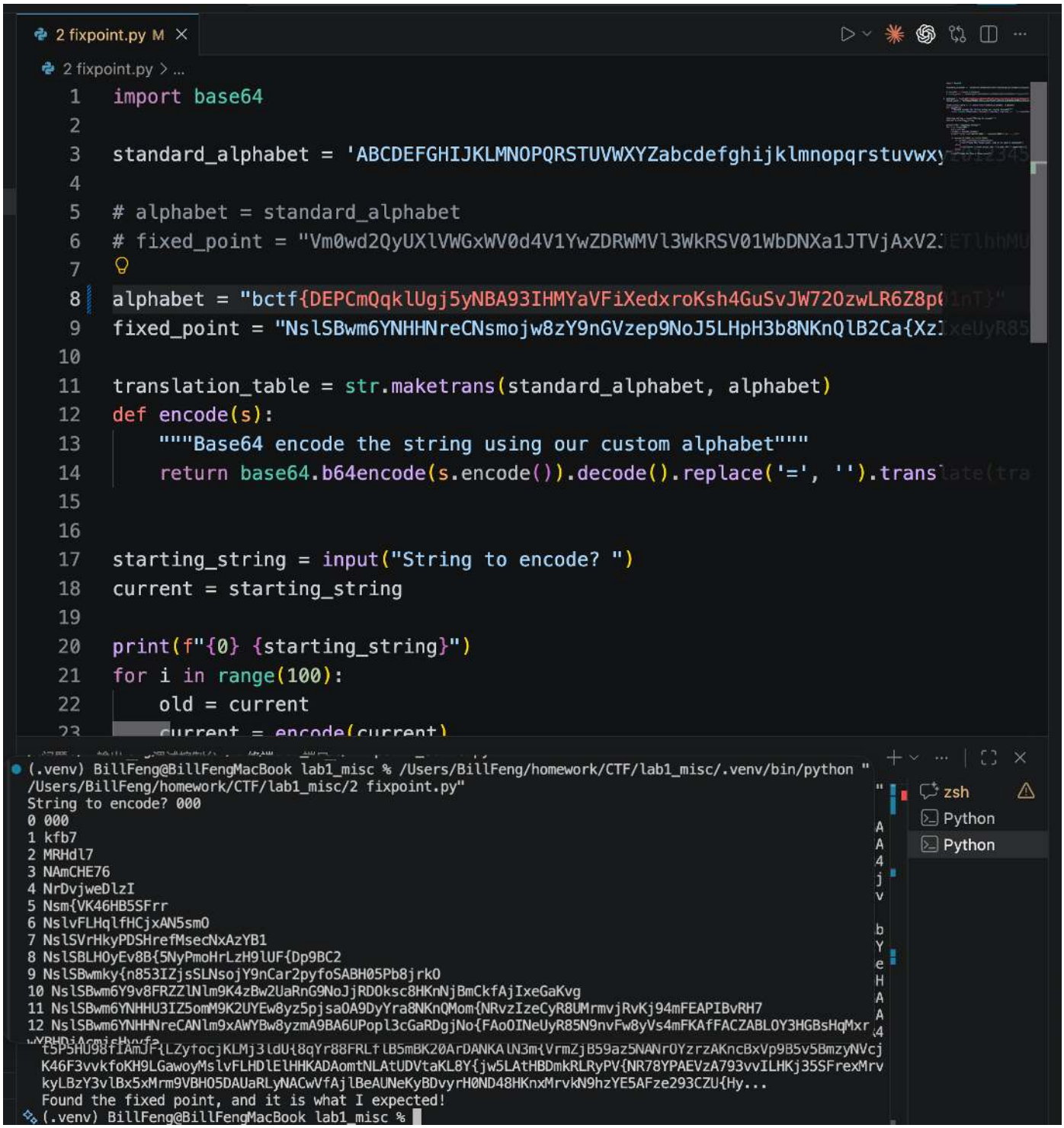
2 fixpoint.py M
2 fixpoint.py > ...
import base64
standard_alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwx
# alphabet = standard_alphabet
# fixed_point = "Vm0wd2QyUXlVWGxWV0d4V1YwZDRWMVl3WkRSV01WbDNXa1JTVjAxV2
alphabet = "bctf{DEPCmQqklUgj5yNBA93IHMYaVFiXedxroKsh4GuSvJW72OzwLR6Z8p
fixed_point = "NslSBwm6YNHHNreCNsmojw8zY9nGVzep9NoJ5LHpH3b8NKnQlB2Ca{Xz
translation_table = str.maketrans(standard_alphabet, alphabet)
def encode(s):
"""Base64 encode the string using our custom alphabet"""
return base64.b64encode(s.encode()).decode().replace('=', '').trans
starting_string = input("String to encode? ")
current = starting_string
print(f"{0} {starting_string}")
for i in range(100):
old = current
current = encode(current)
(.venv) BillFeng@BillFengMacBook lab1_misc % /Users/BillFeng/homework/CTF/lab1_misc/.venv/bin/python "
/Users/BillFeng/homework/CTF/lab1_misc/2 fixpoint.py"
String to encode? 000
0 000
1 kfb7
2 MRHdl7
3 NAmCHE76
4 NrDvjweDlzI
5 Nsm{VK46HB5SFrr
6 NslvFLHqlfHCjxAN5sm0
7 NslSVrHkyPDSHrefMsecNxAzYB1
8 NslSBLHOyEv8B{5NyPmoHrLzH9lUF{Dp9BC2
9 NslSBwmky{n853IZjsSLNsojY9nCar2pyfoSABH05Pb8jrkO
10 NslSBwm6Y9v8FRZZlNlm9K4zBw2UaRnG9NoJjRDOksc8HKnNjBmCkfAjIxeGaKvg
11 NslSBwm6YNHHU3IZ5omM9K2UYEw8yz5pjsaOA9DyYra8NKnQMom{NRvzIzeCyR8UMrmvjRvKj94mFEAPIBvRH7
12 NslSBwm6YNHHNreCANlm9xAWYBw8yzmA9BA6UPopl3cGaRDgjNo{FAoOINeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
t5P5HU98fIAmJF{LZyfocjKLMj3ldU{8qYr88FRLflB5mBK20ArDANKAlN3m{VrmZjB59az5NANrOYzrzAKncBxVp9B5v5BmzyNVcj
K46F3vvkfoKH9LGawoyMslvFLHDlElHHKADAomtNLAtUDVtaKL8Y{jw5LAtHBDmkRLRyPV{NR78YPAEVzA793vvILHKj35SFrexMrv
kyLBzY3vlBx5xMrm9VBHO5DAUaRLyNACwVfAjlBeAUNeKyBDvyrH0ND48HKnxMrvkN9hzYE5AFze293CZU{Hy...
Found the fixed point, and it is what I expected!
(.venv) BillFeng@BillFengMacBook lab1_misc %
zsh
Python
Python

Image 6

3. Task 3 (25%)

3.1 快? 慢?

附件为 `3 slow-login.pcap`。使用 Wireshark 打开后，先观察 HTTP 登录流量。截图中可以看到大量 `POST /login` 请求，响应都是 `HTTP/1.1 200 OK`，但请求到响应的时间差明显分成两类：

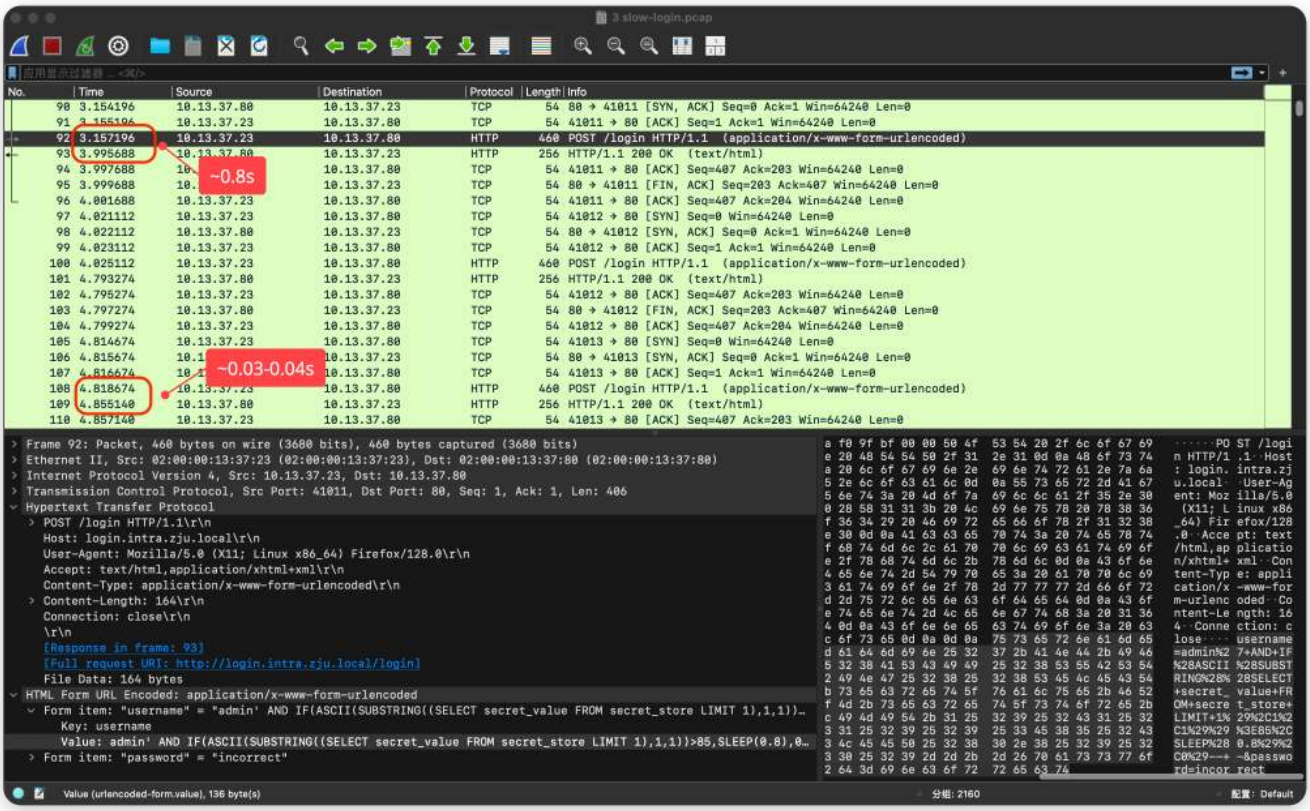

Image 7

- Frame 92 是登录请求，Frame 93 是对应响应，时间差约为 `3.995688 - 3.157196 = 0.838492s`。
- Frame 108 是登录请求，Frame 109 是对应响应，时间差约为 `4.855140 - 4.818674 = 0.036466s`。

选中 Frame 92 后，在下方的 `HTML Form URL Encoded` 中可以看到请求参数：

```
username=admin' AND IF(ASCII(SUBSTRING((SELECT secret_value FROM secret_store LIMIT 1),1,1))>85,SLEEP(0.8),0)-- -
password=incorrect
```

这说明攻击者在做 SQL 时间盲注：如果条件成立，服务器执行 `SLEEP(0.8)`，响应就会变慢；如果条件不成立，响应会很快返回。由于响应正文都是相同的 `Invalid credentials`，所以不能看内容判断真假，只能看请求与响应之间的时间差。

3.2 异常请求&响应

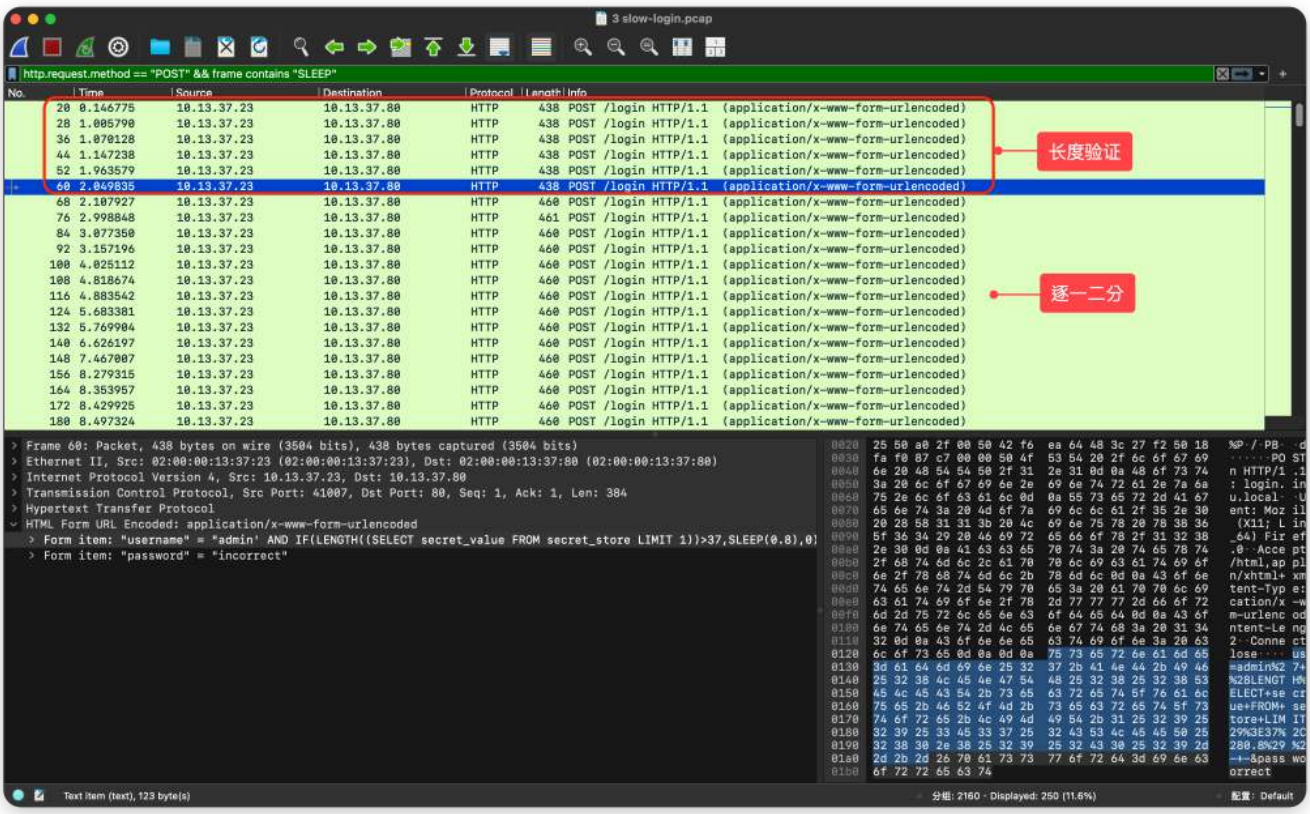

Image 8

一开始我先在 Wireshark 的显示过滤器中只看登录请求：

```
http.request.method == "POST" && http.request.uri == "/login"
```

因为前面已经在表单参数中看到了 `SLEEP(0.8)`，所以进一步用下面的过滤器只保留可疑请求：

```
http.request.method == "POST" && frame contains "SLEEP"
```

过滤后可以看到请求大致分成两段：前面几条是在判断 `secret_value` 的长度，后面大量请求是在逐个字符二分。截图中选中的请求就是长度判断：

```
username=admin' AND IF(LENGTH((SELECT secret_value FROM secret_store LIMIT
1))>37,SLEEP(0.8),0)-- -
```

每个登录请求都会新建一条 TCP 连接。以截图为例，Frame 92 是请求，Wireshark 在详情中提示它的响应是 Frame 93，因此可以用 Frame 93 的时间减去 Frame 92 的时间，得到这次请求的响应延迟。

普通失败登录约 `0.03s ~ 0.06s` 返回；条件为真的盲注请求会执行 `SLEEP(0.8)`，响应延迟约 `0.77s ~ 0.87s`。因此取 `0.4s` 作为阈值：

```
delay > 0.4s  => 条件为真
delay <= 0.4s => 条件为假
```

3.3 恢复

攻击者的二分查询过程已经完整记录在 pcap 里，所以我只需要把每条条件的真假整理出来。先看长度判断请求：

条件	延迟	结果
LENGTH(...) > 32	0.833s	真
LENGTH(...) > 36	0.791s	真
LENGTH(...) > 37	0.032s	假
LENGTH(...) > 38	0.060s	假
LENGTH(...) > 40	0.046s	假
LENGTH(...) > 48	0.033s	假

Table 2

由上表可知，长度大于 `36` 为真，但大于 `37` 为假，所以长度为 `37`。

随后每个字符都用类似下面的条件继续二分：

```
ASCII(SUBSTRING((SELECT secret_value FROM secret_store LIMIT 1),pos,1)) > mid
```

以第 1 个字符为例，流量里有：

条件	延迟	结果
ASCII(...,1,1) > 89	0.775s	真
ASCII(...,1,1) > 90	0.036s	假

Table 3

因此第 1 个字符 ASCII 大于 `89`，但不大于 `90`，所以它就是 `90`，对应字符 `Z`。

3.4 python脚本

手工算一两个字符可以验证思路，但一共有几百个登录请求。逐条整理比较容易出错。

脚本完成的工作是：

1. 解析 pcap 中的 Ethernet、IPv4、TCP 和 HTTP 明文数据。
2. 按同一 TCP 连接把 `POST /login` 请求和 `HTTP/1.1 200 OK` 响应配对。
3. 计算响应延迟，并用 `0.4s` 阈值判断 SQL 条件真假。
4. 根据 `LENGTH(...) > mid` 和 `ASCII(SUBSTRING(...)) > mid` 的真假结果还原 secret。

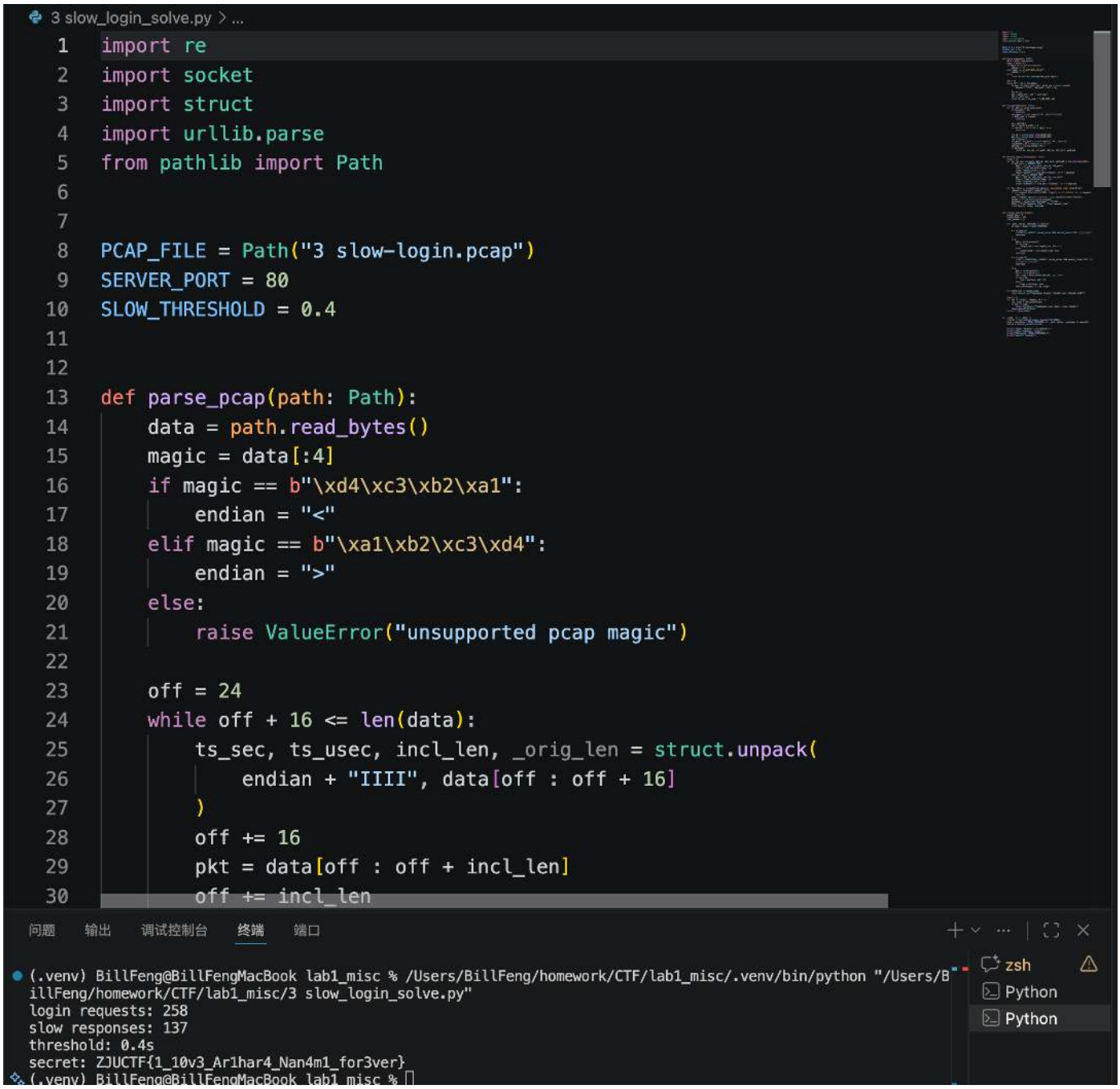

Image 9

`secret: ZJUCTF{1_10v3_Ar1har4_Nan4m1_for3ver}`

4. Task 4 (25%)

4.1 ttl_tells_a_tale

打开后先看协议统计，发现整个流量非常单一，一共 `220` 个 IPv4 包，全部都是 ICMP，其中 `110` 个 echo request 和 `110` 个 echo reply。

也就是说这题重点不在 HTTP/TCP 内容，而是在 ICMP 或 IP 字段里。

发现大部分 echo request 的 TTL 都是 `64`，但是中间夹着一些 TTL 明显不一样的请求。

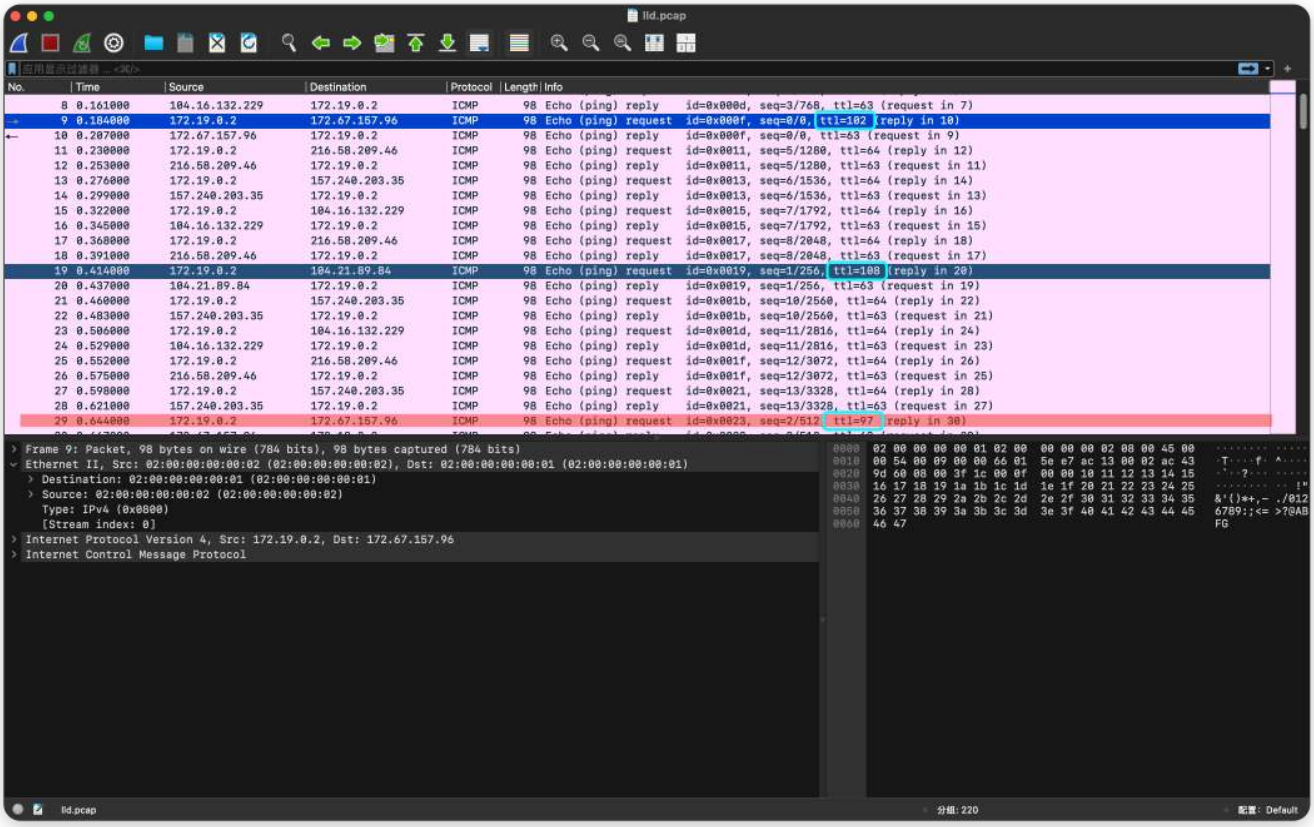

Image 10

截图中普通请求的 TTL 是 `64`，而被标出的异常请求分别是：

```
Frame 9:  ttl=102
Frame 19: ttl=108
Frame 29: ttl=97
```

这些数值刚好落在可打印 ASCII 范围内，并且：

```
102 -> f
108 -> l
97  -> a
```

开头正好是 `fla`，所以可以判断 flag 被藏在 ICMP echo request 的 `ip.ttl` 字段里。于是进一步过滤异常 TTL：

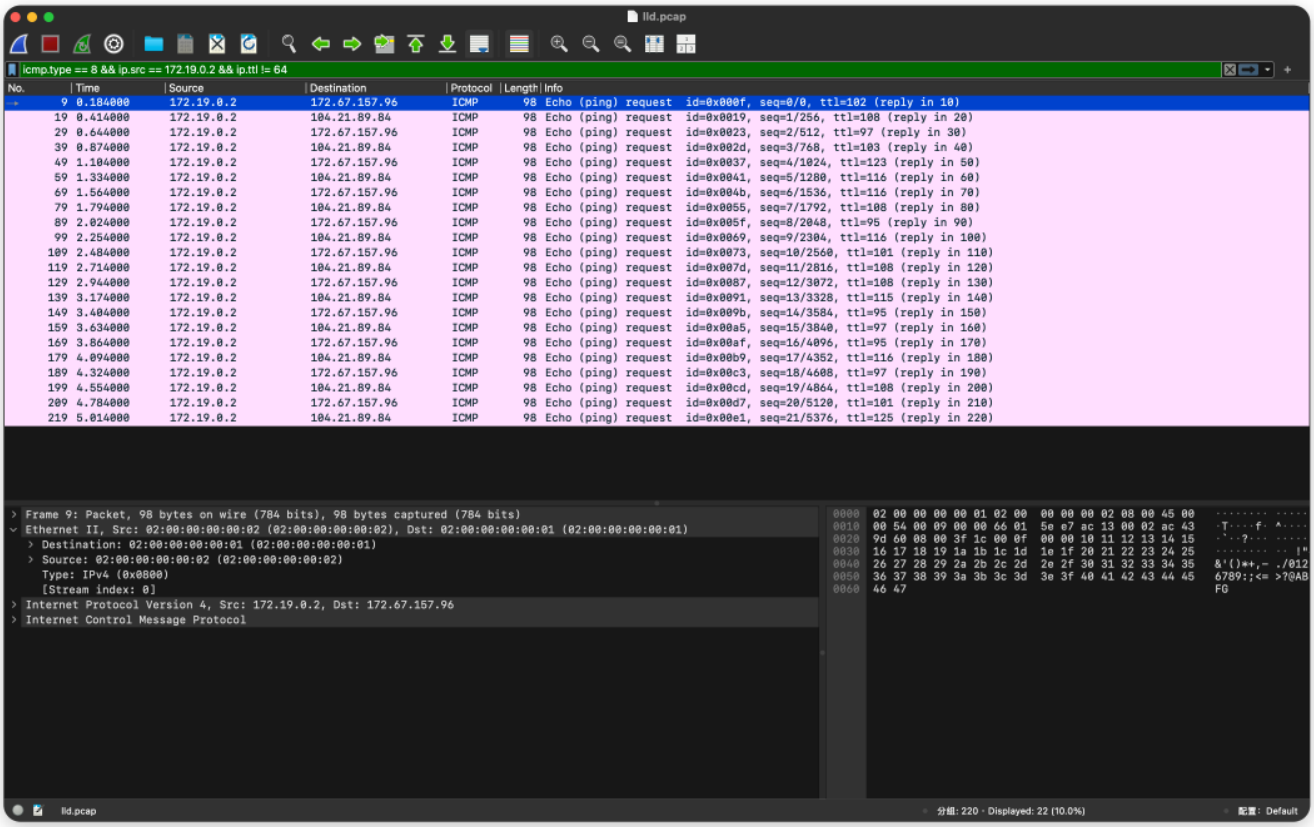

Image 11

使用的过滤器是：

```
icmp.type == 8 && ip.src == 172.19.0.2 && ip.ttl != 64
```

新图中已经把全部异常 TTL 都过滤出来了，共 `22` 个包。按时间顺序读取 `Info` 列中的 `ttl` 值：

```
102 108 97 103 123 116 116 108 95 116 101 108 108 115 95 97 95 116 97 108 101 125
```

将这些十进制数转成 ASCII 即可得到完整 flag：

`flag{ttl_tells_a_tale}`

4.2 简要的python脚本验证

（实则可以直接手搓答案）

不过为了作为附件和验证，写了脚本 `4 lld_solve.py`。

只保留从 `172.19.0.2` 发出的 ICMP echo request，筛出 `ip.ttl != 64`，再把 TTL 按时间顺序转成 ASCII。

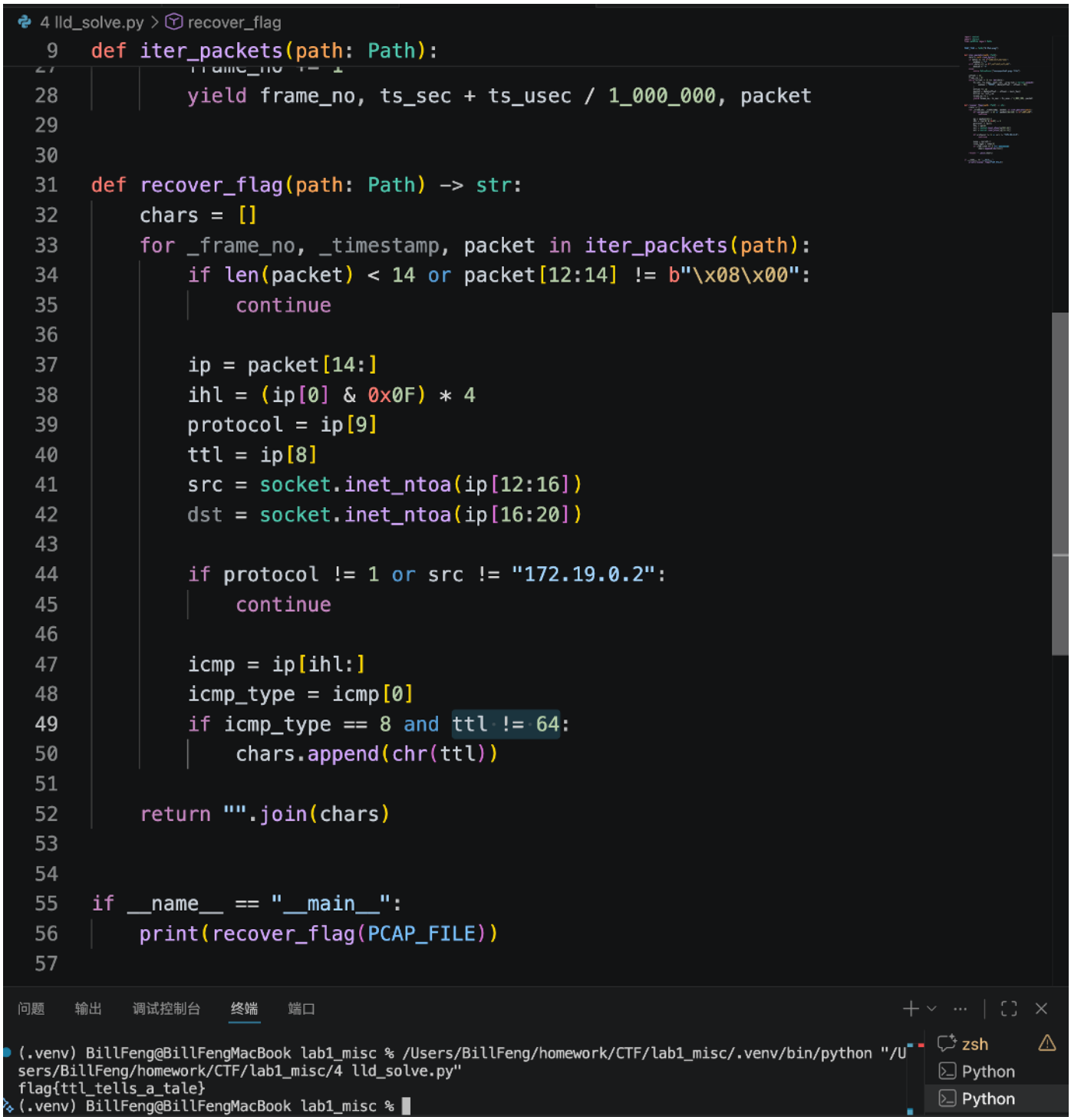

Image 12

`flag{ttl_tells_a_tale}`