

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`：

```

BillFeng@BillFengMacBook lab1_misc % cat -et calculator.lua
$
$
$
^I    ^I^I$
^I$
^I    ^I    ^I$
^I$
^I    ^I    ^I    $
^I$
^I^I^I^I    ^I^I$
^I$
^I^I    ^I    $
^I$
^I^I    ^I^I    $
^I$
^I^I^I    ^I    $
^I$
^I^I    ^I    $
^I$
^I^I    ^I    $
^I$
^I^I    ^I    $
^I$
^I^I    ^I$
^I$
^I^I    ^I    ^I$
^I$
^I    ^I^I    $
^I$
^I^I    ^I    $
^I$
^I^I^I^I^I    ^I$
^I$
$
$
$
$
$
-- nope no flag here, keep looking :P%

```

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

The image shows a code editor with a Python script named `calculator_solve.py` and its execution output in a terminal window.

```

1  #!/usr/bin/env python3
2  with open('calculator.lua', 'rb') as f:
3      lines = f.read().split(b'\n')
4
5  flag = ''
6  for line in lines:
7      if line.strip():
8          continue
9      if len(line) == 0:
10         continue
11     bits = ''.join('1' if b == 9 else '0' for b in line if b in (9, 32))
12     if len(bits) <= 1:
13         continue
14     flag += chr(int(bits[-7:], 2))
15
16 print(flag)
17

```

The terminal output shows the command being run and the resulting flag:

```

(.venv) BillFeng@BillFengMacBook lab1_misc % /Users/BillFeng/homework/CTF/lab1_misc/.venv/bin
/python /Users/BillFeng/homework/CTF/lab1_misc/calculator_solve.py
CIT{hft4bT0415Lb}
(.venv) BillFeng@BillFengMacBook lab1_misc %

```

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:

base64为什么会趋向稳定不动点? 其他base的原理是怎么样的?

直觉上, Base64 反复编码会“稳定”, 不是因为字符串长度不变, 而是因为前缀逐渐稳定。

严格说, 普通有限字符串不可能真的变成完全不变的字符串, 因为 Base64 后长度大约乘 $4/3$ 。题里说的 fixpoint 更像是一个无限字符串的不动点: 反复编码后, 前 1 位、前 2 位、前 1000 位.....都会逐渐固定下来。

为什么会稳定

Base64 是 3 bytes $\rightarrow$ 4 个 6-bit 索引:

```
text
24 bit = 6 bit + 6 bit + 6 bit + 6 bit
```

所以输出的第 1 个字符只依赖输入的第 1 个字节的高 6 位。
输出的前 k 个字符, 只依赖输入的大约 $3k/4$ 个字符。

关键点在这里:

```
text
要确定输出前 k 位, 只需要输入前 3k/4 位
```

因为 $3k/4 < k$, 这是一个“向前缀收缩”的过程。于是前缀的依赖范围越来越小, 最后会被最前面的少数字符决定。反复编码后, 前缀会进入一个固定模式。

例如标准 Base64 alphabet 下, 第一个字符的变化大概可以看成:

```
python
next_char = alphabet[ord(char) >> 2]
```

因为第一个 Base64 字符取的是第一个字节的高 6 位。

很多字符都会这样收敛:

```
text
A -> Q -> U -> V -> V
a -> Y -> W -> V -> V
0 -> M -> T -> V -> V
```

所以标准 Base64 反复编码后, 最前面会稳定成 V。继续看第二位、第三位.....就会得到那个著名前缀:

```
text
Vm0wd2QyUXlVWGxwV0d4...
```

CTF 题里的自定义 alphabet

Base64 的 6-bit 索引序列本质没变, 变的是:

```
text
索引 0..63 -> 输出字符
```

标准 Base64 用:

```
text
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
```

题目换成了自定义 alphabet。

如果 fixed point 是 F, 对 F 做普通 Base64 得到的某一位代表索引 V, 而 fixed point 要求输出还是 F[i], 所以有:

```
text
alphabet[v] = F[i]
```

这就是为什么能从 fix point 反推 alphabet。

其他 base 呢

输出后域变更要求

+ 完全访问 5.5 高

Image 4

题目给出自定义 Base64 alphabet（同时也是flag）：

```
bctf{????????????????????FiXed????????????????p0lnT}
```

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?

1	"abc" -> YWJj	# 3 字节，不加 =
2	"ab" -> YWI=	# 2 字节，加 1 个 =
3	"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

The image shows a Python script named `fixpoint_solve.py` and its execution output in a terminal. The script defines a function `load_challenge_constants` that parses an AST and extracts constants for 'alphabet' and 'fixed_point'. It also defines a function `recover_alphabet` that recovers the alphabet from the fixed point. The terminal output shows the recovered alphabet and a verification step.

```
1 import ast
2 import base64
3 from pathlib import Path
4
5
6 STANDARD_ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789-_"
7 CHALLENGE_FILE = Path("2 fixpoint.py")
8
9
10 def load_challenge_constants(path: Path) -> tuple[str, str]:
11     tree = ast.parse(path.read_text())
12     constants = {}
13     for node in tree.body:
14         if not isinstance(node, ast.Assign):
15             continue
16         for target in node.targets:
17             if isinstance(target, ast.Name) and target.id in {"alphabet", "fixed_point"}:
18                 constants[target.id] = ast.literal_eval(node.value)
19     return constants["alphabet"], constants["fixed_point"]
20
21
22 def recover_alphabet(template: str, fixed_point: str) -> str:
23     recovered = ["?"] * 64
24
25     encoded = base64.b64encode(fixed_point.encode()).decode().rstrip("=")
26     for pos, (std_ch, custom_ch) in enumerate(zip(encoded, fixed_point)):
27         idx = STANDARD_ALPHABET.index(std_ch)
28         if recovered[idx] not in {"?", custom_ch}:
29             raise ValueError(
30                 f"Conflict at output position {pos}: {std_ch} vs {custom_ch}"
31             )
32         recovered[idx] = custom_ch
33
34     return "".join(recovered)
35
36
37 if __name__ == "__main__":
38     alphabet, fixed_point = load_challenge_constants(CHALLENGE_FILE)
39     recovered_alphabet = recover_alphabet(alphabet, fixed_point)
40     print(f"Recovered alphabet (64/64): {recovered_alphabet}")
41
42     # Verification
43     encode(fixed_point).startswith(fixed_point) == True
44     print(f"Verification: encode(fixed_point).startswith(fixed_point) == True")
```

Terminal Output:

```
(.venv) BillFeng@BillFengMacBook lab1_misc % /Users/BillFeng/homework/CTF/lab1_misc/.venv/bin/python /Users/BillFeng/homework/CTF/lab1_misc/fixpoint_solve.py
Recovered alphabet (64/64):
bctf{DEPCmQqkLUgj5yNBA93IHMYaVfiXedxroKsh4GuSvJW720zwLR6Z8p01nT}

Verification:
encode(fixed_point).startswith(fixed_point) == True
(.venv) BillFeng@BillFengMacBook lab1_misc %
```

Chat Interface (Claude Code):

你好像把标准 Base64 字符换成 6-bit 索引:

```
text
T -> 19
n -> 39
N -> 13
s -> 44
U -> 20
0 -> 52
I -> 8
```

而 fixed point 要求编码后前几位仍然是:

```
text
Ns1SB...
```

所以对应关系是:

```
text
alphabet[19] = N
alphabet[39] = s
alphabet[13] = l
alphabet[44] = S
alphabet[20] = B
```

这里只用前 5 位 'Ns1SB'，因此先能推出这 5 条映射。

Image 5

通过脚本实现上述过程。最后验证。

```

2 fixpoint.py M X
2 fixpoint.py > ...
1  import base64
2
3  standard_alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789-._~'
4
5  # alphabet = standard_alphabet
6  # fixed_point = "Vm0wd2QyUXlVWGxwV0d4V1YwZDRwMVl3WkRSV01WbDNxa1JTVjAxV2JETlhhMU"
7  Q
8  alphabet = "bctf{DEPCmQqklUgj5yNBA93IHMYaVFiXedxroKsh4GuSvJW720zwLR6Z8p0InT}"
9  fixed_point = "NslSBwm6YNHHNreCNsmojw8zY9nGVzep9NoJ5LHpH3b8NKnQLB2Ca{XzIxeUyR85"
10
11  translation_table = str.maketrans(standard_alphabet, alphabet)
12  def encode(s):
13      """Base64 encode the string using our custom alphabet"""
14      return base64.b64encode(s.encode()).decode().replace('=', '').translate(translation_table)
15
16
17  starting_string = input("String to encode? ")
18  current = starting_string
19
20  print(f"{0} {starting_string}")
21  for i in range(100):
22      old = current
23      current = encode(current)
24
25  (.venv) BillFeng@BillFengMacBook lab1_misc % /Users/BillFeng/homework/CTF/lab1_misc/.venv/bin/python "
26  /Users/BillFeng/homework/CTF/lab1_misc/2 fixpoint.py"
27  String to encode? 000
28  0 000
29  1 kfb7
30  2 MRHd17
31  3 NAmCHE76
32  4 NrDvjeDlzI
33  5 Nsm{VK46HB5SFrr
34  6 NslvFLHqlfHCjxAN5sm0
35  7 NslSVrHkyPDSHrefMsecNxAYB1
36  8 NslSBLH0yEv8B{5NyPmoHrLzH9lUF{Dp9BC2
37  9 NslSBwmky{n853IZjsSLNsojY9nCar2pyfoSABH05Pb8jrk0
38  10 NslSBwm6Y9v8FRZZlNm9K4zBw2UaRnG9NoJjRD0ksc8HKnNjBmCkfAjIxeGaKvg
39  11 NslSBwm6YNHHU3IZ5omM9K2UYEw8yz5pjsa0A9DyYra8NKnQMom{NRvzIzeCyR8UMrmvjRvKj94mFEAPIBvRH7
40  12 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
41  13 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
42  14 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
43  15 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
44  16 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
45  17 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
46  18 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
47  19 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
48  20 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
49  21 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
50  22 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
51  23 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
52  24 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
53  25 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
54  26 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
55  27 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
56  28 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
57  29 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
58  30 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
59  31 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
60  32 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
61  33 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
62  34 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
63  35 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
64  36 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
65  37 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
66  38 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
67  39 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
68  40 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
69  41 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
70  42 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
71  43 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
72  44 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
73  45 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
74  46 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
75  47 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
76  48 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
77  49 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
78  50 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
79  51 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
80  52 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
81  53 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
82  54 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
83  55 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
84  56 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
85  57 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
86  58 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
87  59 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
88  60 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
89  61 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
90  62 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
91  63 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
92  64 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
93  65 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
94  66 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
95  67 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
96  68 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
97  69 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
98  70 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
99  71 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
100 72 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
101 73 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
102 74 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
103 75 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
104 76 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
105 77 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
106 78 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
107 79 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
108 80 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
109 81 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
110 82 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
111 83 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
112 84 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
113 85 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
114 86 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
115 87 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
116 88 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
117 89 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
118 90 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
119 91 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
120 92 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
121 93 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
122 94 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
123 95 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
124 96 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
125 97 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
126 98 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
127 99 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
128 100 NslSBwm6YNHHNreCANlm9xAlwYBw8yzmA9BA6UPop13cGaRdgjNo{FAo0INeUyR85N9nvFw8yVs4mFKAfFACZABLOY3HGBsHqMxr
Found the fixed point, and it is what I expected!
(.venv) BillFeng@BillFengMacBook lab1_misc %

```

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:

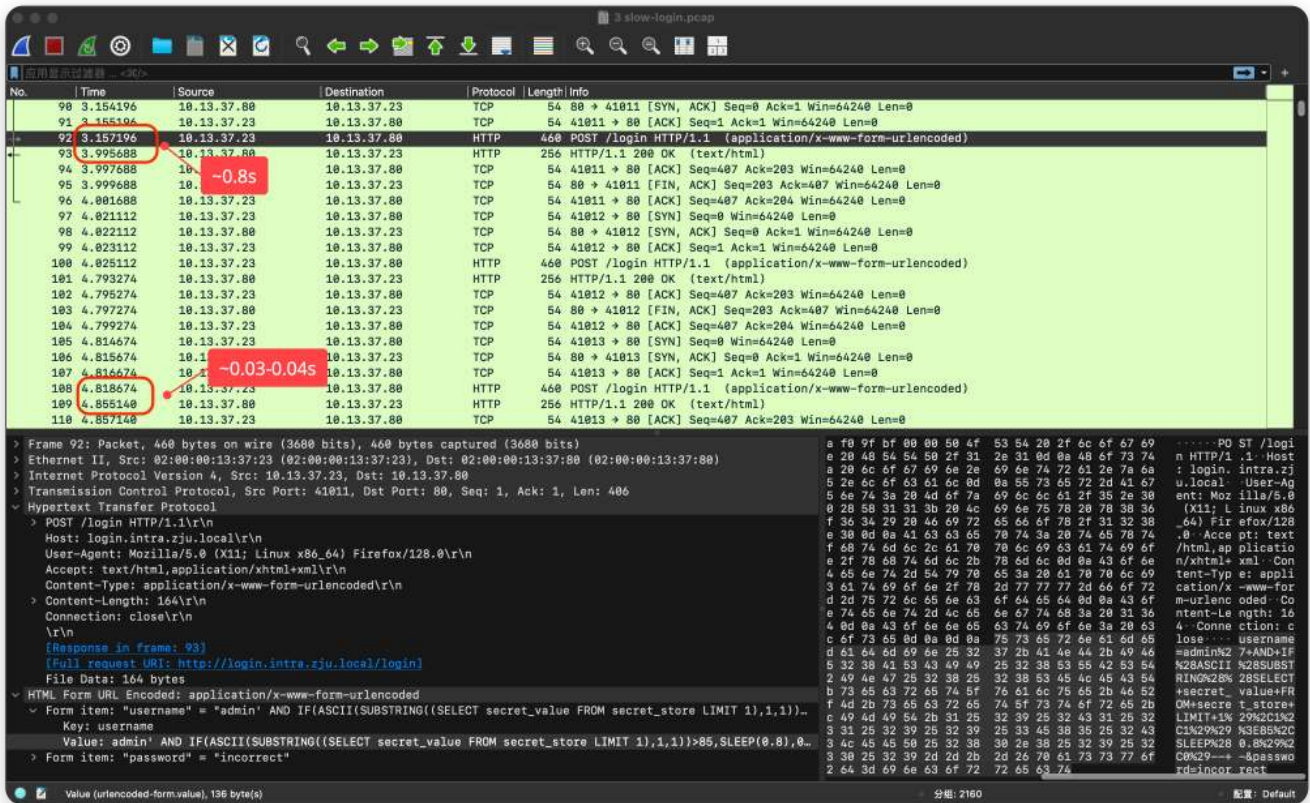


Image 7

- 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:

```
1 username=admin' AND IF(ASCII(SUBSTRING((SELECT secret_value FROM secret_store LIMIT 1),1,1))>85,SLEEP(0.8),0) --
2 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

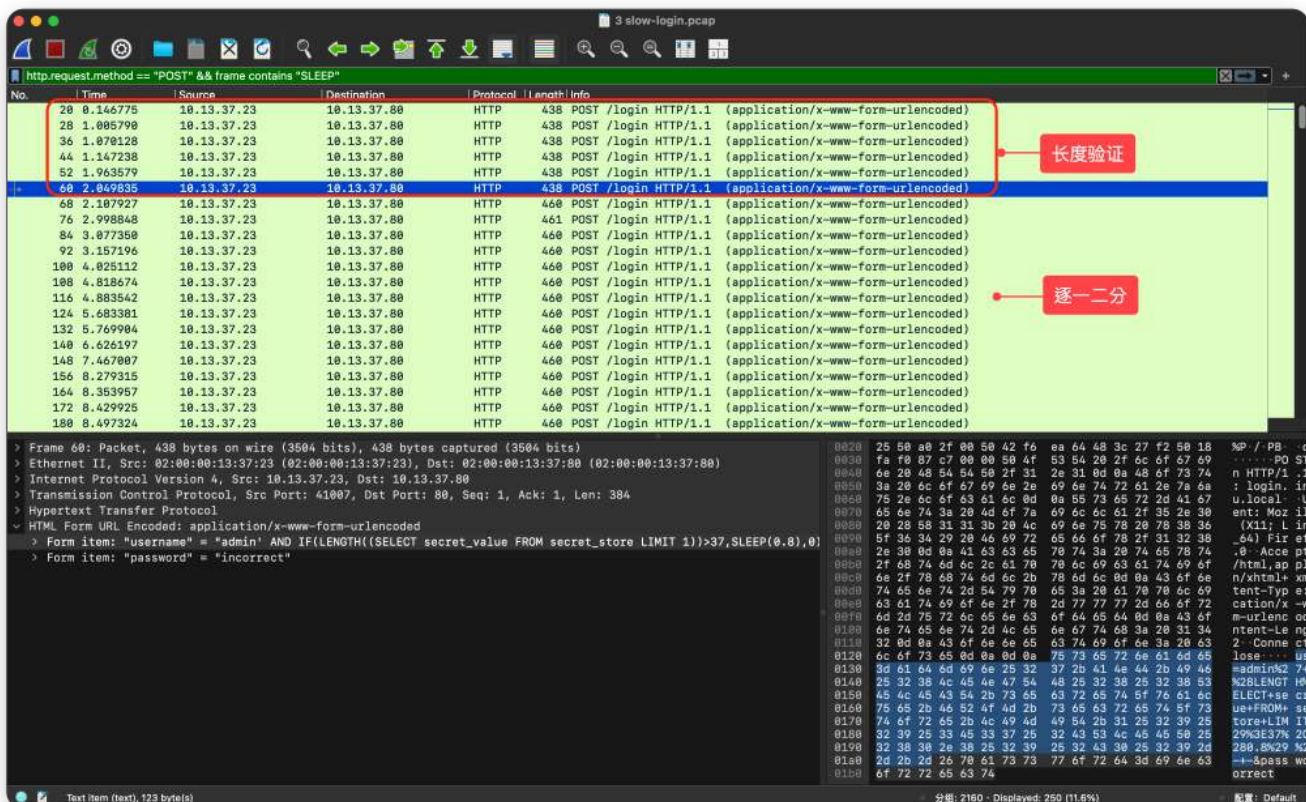


Image 8

一开始我先在 Wireshark 的显示过滤器中只看登录请求：

```
1 | 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：

```
1 | 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：

```
1 | 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：

```
1 | delay > 0.4s => 条件为真
2 | 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：

```
1 | 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。

```
3 slow_login_solve.py > ...
1 import re
2 import socket
3 import struct
4 import urllib.parse
5 from pathlib import Path
6
7
8 PCAP_FILE = Path("3 slow-login.pcap")
9 SERVER_PORT = 80
10 SLOW_THRESHOLD = 0.4
11
12
13 def parse_pcap(path: Path):
14     data = path.read_bytes()
15     magic = data[:4]
16     if magic == b"\xd4\xc3\xb2\xa1":
17         endian = "<"
18     elif magic == b"\xa1\xb2\xc3\xd4":
19         endian = ">"
20     else:
21         raise ValueError("unsupported pcap magic")
22
23     off = 24
24     while off + 16 <= len(data):
25         ts_sec, ts_usec, incl_len, _orig_len = struct.unpack(
26             endian + "IIII", data[off : off + 16]
27         )
28         off += 16
29         pkt = data[off : off + incl_len]
30         off += incl_len
```

问题 输出 调试控制台 终端 端口

```
(.venv) BillFeng@BillFengMacBook lab1_misc % /Users/BillFeng/homework/CTF/lab1_misc/.venv/bin/python "/Users/BillFeng/homework/CTF/lab1_misc/3 slow_login_solve.py"
login requests: 258
slow responses: 137
threshold: 0.4s
secret: ZJUCTF{1_10v3_Ar1har4_Nan4m1_for3ver}
(.venv) BillFeng@BillFengMacBook lab1_misc %
```

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。

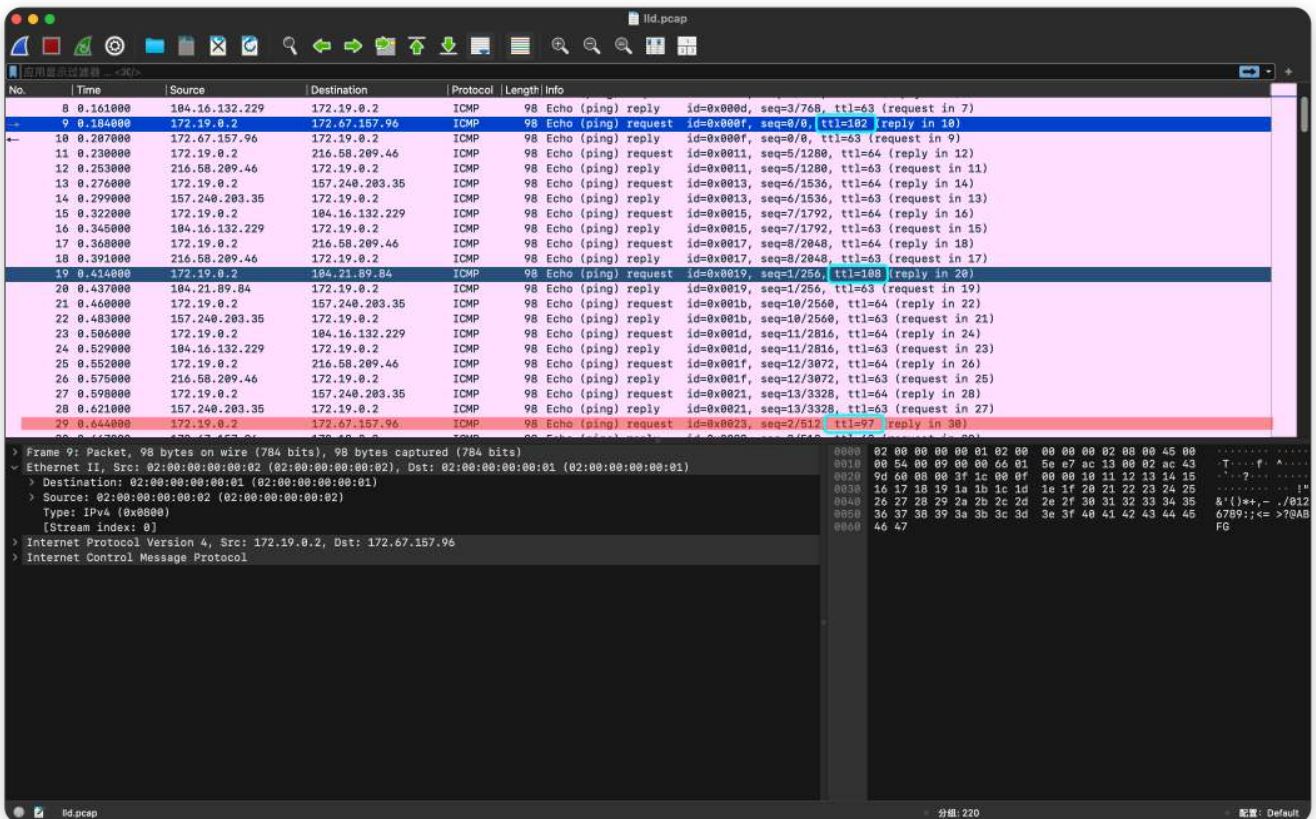


Image 10

截图中普通请求的 TTL 是 64，而被标出的异常请求分别是：

- 1 | Frame 9: ttl=102
- 2 | Frame 19: ttl=108
- 3 | Frame 29: ttl=97

这些数值刚好落在可打印 ASCII 范围内，并且：

- 1 | 102 -> f
- 2 | 108 -> l
- 3 | 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：

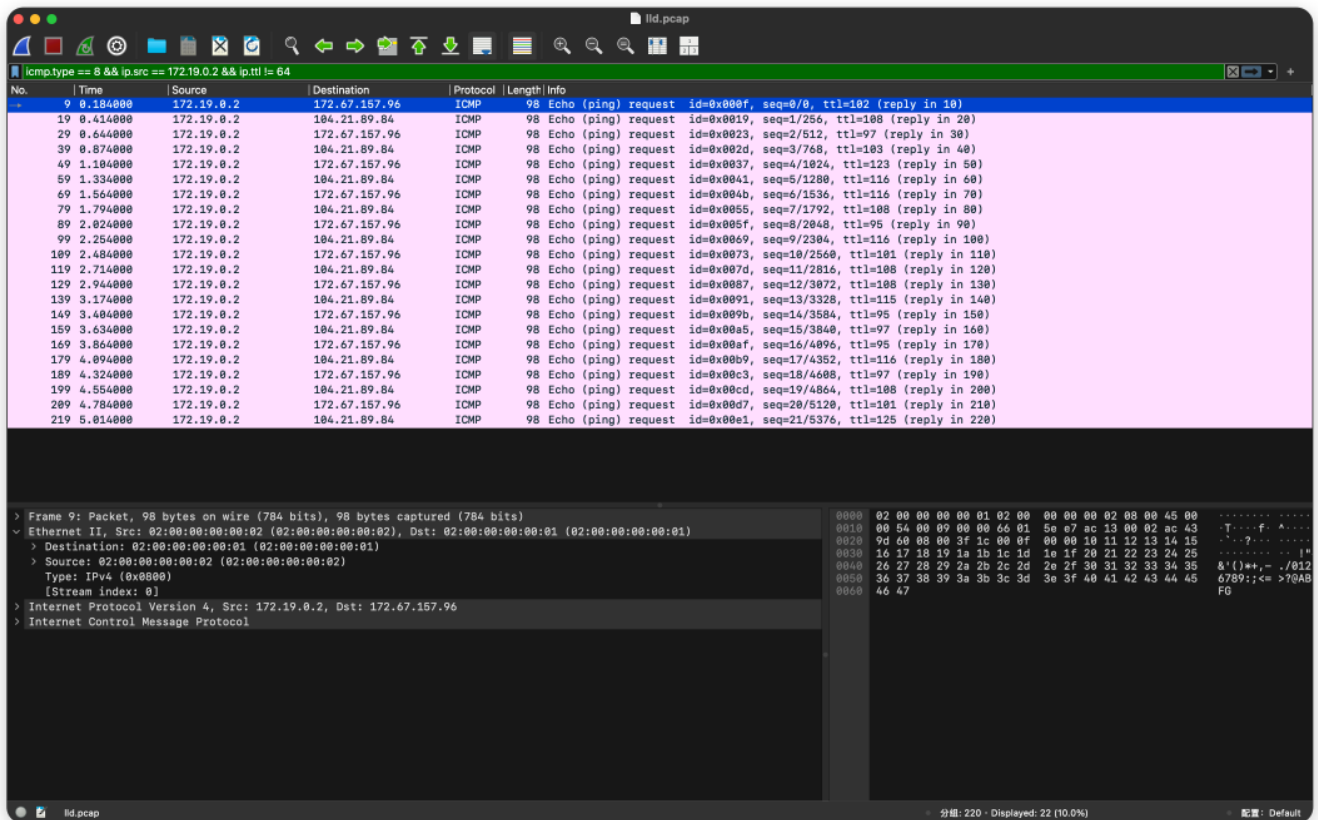


Image 11

使用的过滤器是：

```
1 | 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：

```
1 | 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。

```
4 lld_solve.py > recover_flag
9 def iter_packets(path: Path):
27     frame_no = 1
28     yield frame_no, ts_sec + ts_usec / 1_000_000, packet
29
30
31 def recover_flag(path: Path) -> str:
32     chars = []
33     for _frame_no, _timestamp, packet in iter_packets(path):
34         if len(packet) < 14 or packet[12:14] != b"\x08\x00":
35             continue
36
37         ip = packet[14:]
38         ihl = (ip[0] & 0x0F) * 4
39         protocol = ip[9]
40         ttl = ip[8]
41         src = socket.inet_ntoa(ip[12:16])
42         dst = socket.inet_ntoa(ip[16:20])
43
44         if protocol != 1 or src != "172.19.0.2":
45             continue
46
47         icmp = ip[ihl:]
48         icmp_type = icmp[0]
49         if icmp_type == 8 and ttl != 64:
50             chars.append(chr(ttl))
51
52     return "".join(chars)
53
54
55 if __name__ == "__main__":
56     print(recover_flag(PCAP_FILE))
57
```

问题 输出 调试控制台 终端 端口

(.venv) BillFeng@BillFengMacBook lab1_misc % /Users/BillFeng/homework/CTF/lab1_misc/.venv/bin/python "/Users/BillFeng/homework/CTF/lab1_misc/4 lld_solve.py"
flag{ttl_tells_a_tale}

(.venv) BillFeng@BillFengMacBook lab1_misc %

Image 12

flag{ttl_tells_a_tale}