解题报告

BY 魏铭岑

昵称: sAy');DROP TABL p;-- （本来想用sAy');DROP TABLE players;--，但昵称有19字符长度限制= =）
Email: mingcenwei@pku.edu.cn

1 →签到←

签到题给的信息`c3ludHtKM3lwYnpyIGdiIDBndSBDWEggVGhUaFRoLCByYXdibCBndXIgdG56ciF9`看起来似乎是**base64**编码，解码为`synt{J3ypbzr gb 0gu CXH ThThTh, rawbl gur tnzr!}`，与`flag{...}`的形式已经非常接近了，`synt`有4位，`flag`恰好也有4位，猜测是恺撒密码，发现的确将所有字母移位13位就得到答案，是**ROT13**密码。

解密脚本见`./1/script.py`。制表符能更语义性地表达缩进的概念，所以我选择用制表符进行缩进，哼（下同）！（不同用户的制表符的长度设定不一样完全不是问题，因为代码下载到本地时，用户应该首先用格式化软件将代码格式化为自己熟悉的样式；把代码上传到公共服务器时，再用格式化软件将代码格式化为团队要求的标准格式。Hard-coding代码样式（比如说4个空格的缩进）是极其糟糕的，就像hard-coding其它magic numbers一样。）

2 主的替代品

既然不能出现`main`字样，就想到用字符串拼接。打印`main`字符串可用`printf("%sain", "m")`，要定义`main`函数则可以先定义宏`#define FUNC(INITIAL) INITIAL ## ain`，然后使用`FUNC(m)`来表示`main`。

代码见`./2/test.c`。（**注**：这里的代码文件与我本来提交的代码文件并不完全相同，我本来提交的代码文件为`./2/test-original.c`。因为比赛时时间比较紧迫，所以当时我写的一些代码的风格不太好，我现在提交的代码是我稍微优化格式一些后的，下同。）

3 小北问答 1202

3.1

*Google*搜索“北京大学 计算中心机房 门牌号”，出来的第一个链接里即有结果。（求最大质因数使用*Maxima*中的`factor`函数即可）。

图 1.

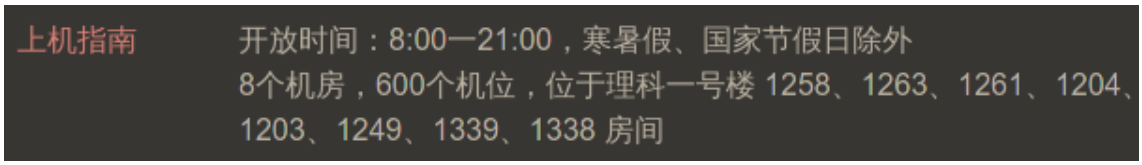


图 2.

3.2

*Google*搜索“"讲得好" "作业少" "考试水" "给分高"”，出来的唯一一个链接里面有答案。

图 3.

图 4.

3.3

*Google*搜索“HTCPCP-TEA protocol”，发现*rfc7168*，文内搜索“coffee”字样即可找到答案。

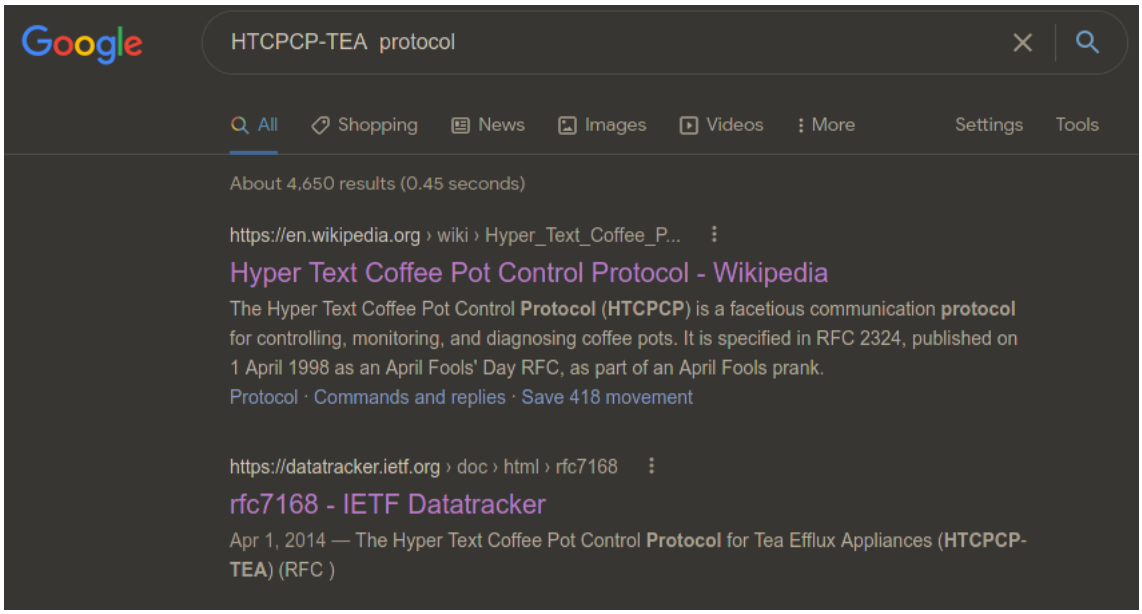


图 5.

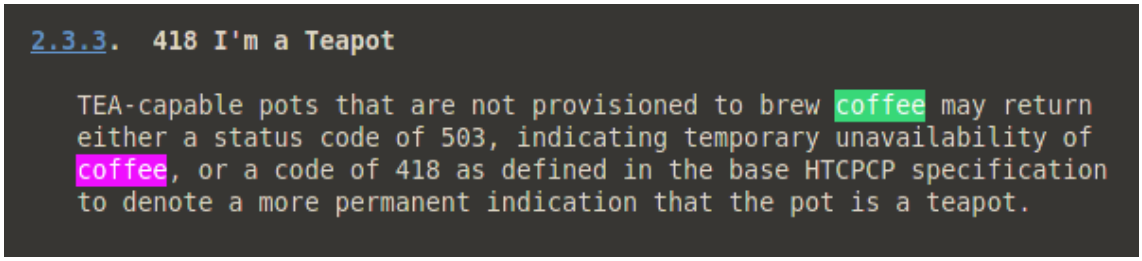


图 6.

3.4

*Google*搜索“Conway's Game of Life stable”，出现的第一个链接中即有答案。

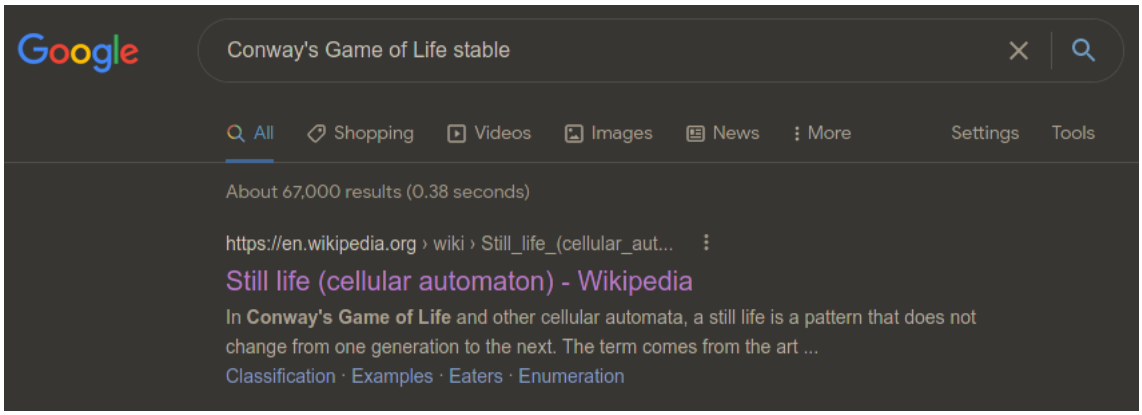
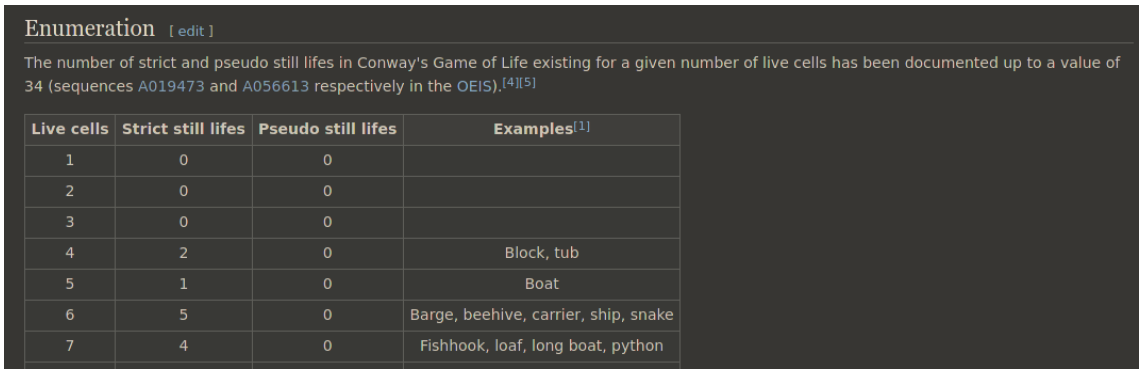


图 7.



Enumeration [edit]

The number of strict and pseudo still lifes in Conway's Game of Life existing for a given number of live cells has been documented up to a value of 34 (sequences A019473 and A056613 respectively in the OEIS).[4][5]

Live cells	Strict still lifes	Pseudo still lifes	Examples[1]
1	0	0	
2	0	0	
3	0	0	
4	2	0	Block, tub
5	1	0	Boat
6	5	0	Barge, beehive, carrier, ship, snake
7	4	0	Fishhook, loaf, long boat, python

图 8.

3.5

*Google*搜索“"FAStT" Management Suite Java default password”，出来的结果中便能找到答案。

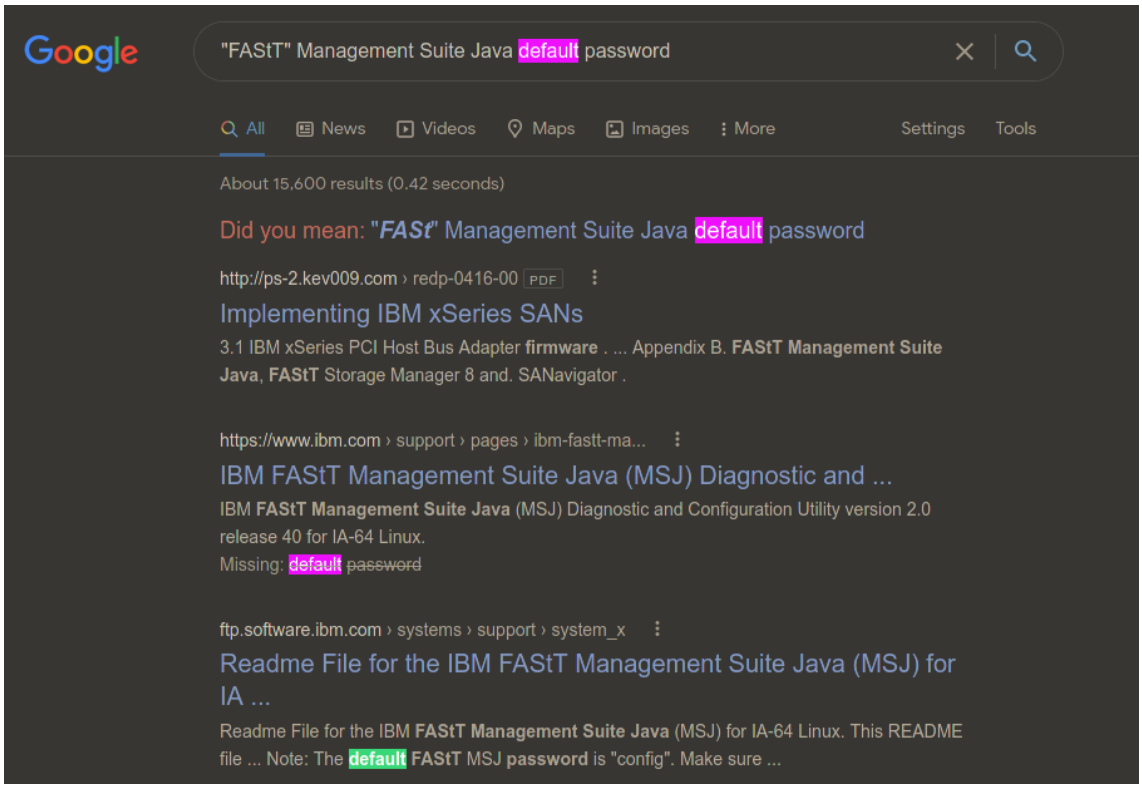


图 9.

3.6

*Google*搜索“最小的汉信码图案”，出来的结果中便能找到答案。

图 10.

3.7

*Google*搜索“国密算法 椭圆曲线密码”，出来的结果中便能找到答案。

图 11.

3.8

*ICANN*一直在`https://data.iana.org/TLD/tlds-alpha-by-domain.txt`更新着最新的可用的TLD列表（参见`https://www.icann.org/resources/pages/tlds-2012-02-25-en`）。使用*Internet Archive*的*Wayback Machine*，访问距离2013年5月4日最接近的2013年5月12日的存档，去

除掉#开头的注释行，然后统计行数即可得到结果。

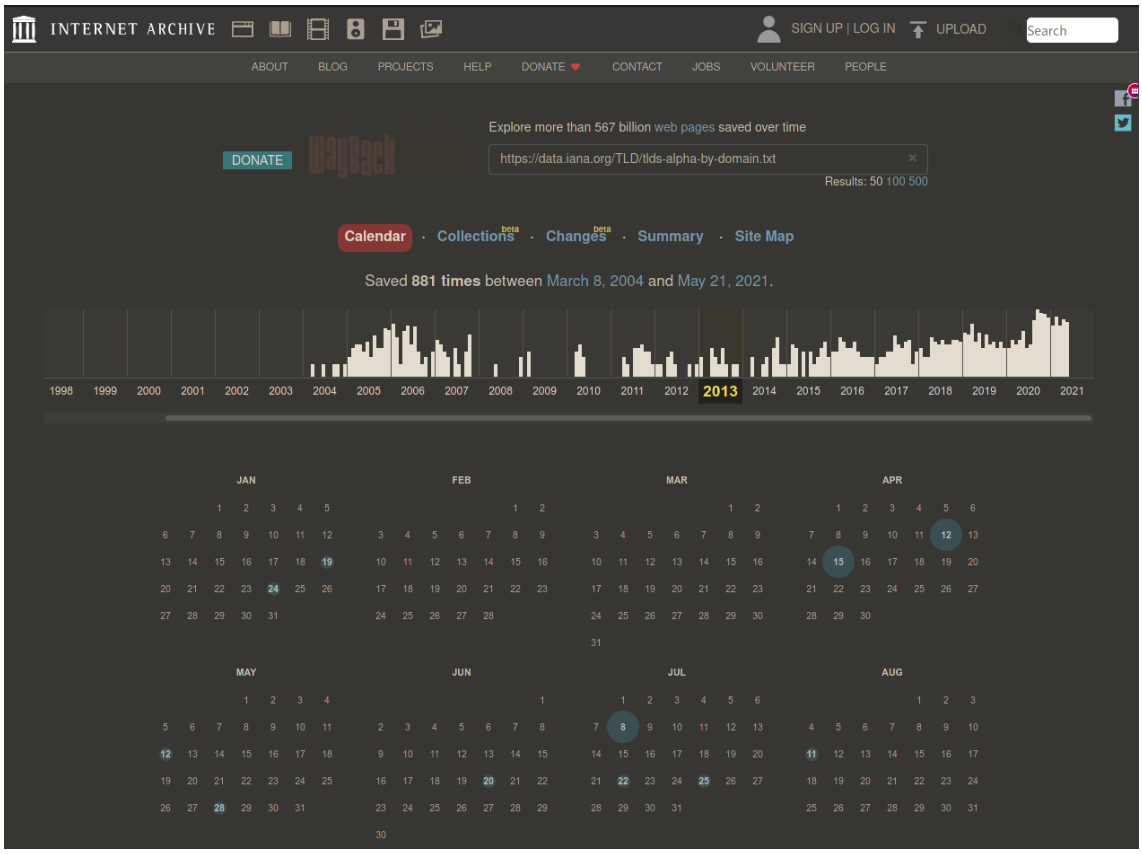


图 12.

图 13.

4 与佛论禅网大会员

尝试使用`unzip`程序解压该图片，发现能解压出来，并且提示前面有236978个多余的字节，因此保留前面的236978个字节为新的**GIF**图片，剩下的字节则为**ZIP**文件。第一个flag没有密码，直接获得。

```
[I] say@Arch-on-Gram ~/C/L/p/g/t> ll
.rw-r--r-- 1 232Ki say say 2021-05-23 19:08 quiz.gif
[I] say@Arch-on-Gram ~/C/L/p/g/t> unzip quiz.gif
Archive:  quiz.gif
warning [quiz.gif]:  236978 extra bytes at beginning or within zipfile
  (attempting to process anyway)
  inflating: flag1.txt
[quiz.gif] flag2.txt password:
```

图 14.

后来在网上查询，又了解到本题的另一种在CTF比赛中似乎比较常规的解法，即直接使用`binwalk`程序查看提供的文件，也可获得第一个flag。

```
[I] say@Arch-on-Gram ~/C/L/p/g/t> ll
.rw-r--r-- 1 232Ki say say 2021-05-23 19:08 quiz.gif
[I] say@Arch-on-Gram ~/C/L/p/g/t> binwalk quiz.gif

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             GIF image data, version "89a", 1034 x 994
236978        0x39DB2         Zip archive data, at least v2.0 to extract, compressed size: 50, uncompressed size: 48, name: flag1.txt
237067        0x39E0B         Zip archive data, encrypted at least v2.0 to extract, compressed size: 60, uncompressed size: 50, name: flag2.txt
237348        0x39F24         End of Zip archive, footer length: 22
```

图 15.

然后使用*ImageMagick*的`convert`程序将**GIF**的每一帧提取出来变成**PNG**文件。

```
[I] say@Arch-on-Gram ~/C/L/p/g/t> ll
.rw-r--r-- 1 232Ki say say 2021-05-23 19:08 quiz.gif
[I] say@Arch-on-Gram ~/C/L/p/g/t> head -c 236978 quiz.gif > new.gif
[I] say@Arch-on-Gram ~/C/L/p/g/t> convert new.gif new.png
[I] say@Arch-on-Gram ~/C/L/p/g/t> ll
.rw-r--r-- 1 194Ki say say 2021-05-23 19:14 new-0.png
.rw-r--r-- 1 3.3Ki say say 2021-05-23 19:14 new-1.png
.rw-r--r-- 1 9.0Ki say say 2021-05-23 19:14 new-2.png
.rw-r--r-- 1 1.5Ki say say 2021-05-23 19:14 new-3.png
.rw-r--r-- 1 5.9Ki say say 2021-05-23 19:14 new-4.png
.rw-r--r-- 1   747 say say 2021-05-23 19:14 new-5.png
.rw-r--r-- 1   542 say say 2021-05-23 19:14 new-6.png
.rw-r--r-- 1   570 say say 2021-05-23 19:14 new-7.png
.rw-r--r-- 1 231Ki say say 2021-05-23 19:14 new.gif
.rw-r--r-- 1 232Ki say say 2021-05-23 19:08 quiz.gif
```

图 16.

接着使用在网上查询了解到的*StegSolve*软件察看每一帧的图片，发现出现时间仅有10ms的偶数帧（不算

背景图片的话则是奇数帧）在red/green/blue plane为比较小的值的时候可能出现类似于二维码的图案。其中有一帧的尺寸需要剪裁（如下图，剪裁掉左边的边缘），并且有的图片要用空白覆盖掉噪音部分（例如下图的左上角）。

图 17.

使用*GIMP*软件进行拼接，得到尺寸为241 × 241的二维码，但还缺少左上角、右上角、左下角的定位区，以及右下角的校准区。取已有二维码的相关区域覆盖上去（覆盖时注意定位区需占7格，校准区需占5格且中心点所在位置是从较近边缘数起第7格），即可扫码得到第二个flag的解压密码（不能直接用微信扫码，那样会直接打开链接。我是用*Android*手机的*Binary Eye* app扫的码）。

图 18.

部分相关文件可见`./4`文件夹。

5 2038 年的银行

这道题先从一个银行存钱再借钱，然后借的钱存入另一个银行，再从另一个银行借钱，如此反复，可以将3个银行的借钱额度都升高到最高的200000...（后面具体有多少个0忘记了）。然后注意到借钱、存钱都会出现类似于溢出的情况，即在有了几次利率或利息过后，值可能变成负数。多睡觉几天，并且调整存款或借款的额度，等待时机，可以使得借款额度变成比现金小的正数，这时候就能用现金还完钱。然后再存钱获取利息（注意不要存太多，否则可能增加利息后溢出变成负数；取钱也不要一次取太多，否则和现金加在一起可能变成负数），然后就能买到flag了。该题应该有比较数学化的严格解法（在研究出具体的溢出临界点和溢出方式后），但我没时间去找，多玩了几次游戏就通过了。

6 人类行为研究实验

浏览器先设置**HTTP**代理为`http://prob01.geekgame.pku.edu.cn:10001/`，即可访问实验室内网。第一个flag，我先是通过玩游戏获取的，策略大概就是出现60000+的数值后认为其是最大的，或者在数字快要出现完的时候如果出现了50000+的数值也认为其是最大的。玩了几次后就通过了。但刚通过，检查网页*JavaScript*代码就发现其实有捷径，在`maximum.js`文件（见`./6/maximum.js`）中有函数`function get(won)`，其中`target`的`&token`前面部分用*JavaScript*算出来其实就是`/flag?k=cyclononane`，故直接访问`http://game.pku.edu.cn/flag?k=cyclononane&token=${TOKEN}`可以快捷获取第一个flag。

至于第二个flag，看`maximum.js`文件中的`submit`函数可知要访问`http://iaaa.pku.edu.cn/?token=${TOKEN}&score=${s}`。然后检查**Cookies**，可以看到**JWT**，例如下面这样的：

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc2FkbWluIjoiZmFsc2UifQ.YZ2tPUJAnqZcHrDPQC--kO6axFmVQkOn1KCA7pzU-KO
```

以“.”作分隔，分别解码**base64url**，上面例子中第一段为`{"typ":"JWT","alg":"HS256"}`，第二段为`{"isadmin":"false"}`（**base64url**末尾应该有两个“=”，被去掉了），然后取前两段（包括中间的“.”）算**HMAC256**，密钥选为空字符串，发现结果的**base64url**恰好就是第三段（去掉末尾的等号）。故可伪造`{"isadmin":"true"}`的**JWT Cookie**。相关代码如下图（使用的*fish* **shell**，下同）：

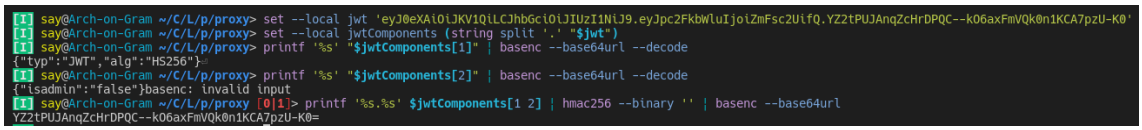


图 19.

同理，在*IAAA*网页登录时，检查到浏览器GET了：

`http://game.pku.edu.cn/callback?_rand=${RANDOM}&jwt=${JWT}&token=${TOKEN}`

其中`jwt`形如：

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpZGVudGl0eSI6InN0dWRlbnQifQ.yY787WIxisr88ADjV-3q2MdvdFoYJvIlA9IChYFvn_Y
```

第二段**base64url**解码为`{"identity":"student"}`（**base64url**末尾的"="被去掉了），故可伪造`{"identity":"teacher"}`的**JWT**，然后就能顺利登录拿到flag了。

7 人生苦短

检查"开放源代码"，发现*flask*开启了debug模式：`app.run('0.0.0.0', 5000, True)`。故可尝试让服务器端出错出现debug界面从而泄漏`app.secret_key`。我们能控制的`action = request.json.get('action', '').strip()`这一行离`app.secret_key`最近，故尝试发送令`action = None`的数据`{"action": null}`（在调用`strip`方法时便会出错）。

代码见`./7/script.js`，安装*npm*的`axios`包后可直接在**shell**中运行`node ./7/script.js | grep -F 'key'`。

8 千年讲堂的方形轮子

第一个**CBC**就用**padding oracle attack**，具体原理可参见https://github.com/neuhalje/aes_padding_oracle_attack。我先用写的`./8/crack-cbc.fish`的代码（代码中我用"byte number"来表示一个字节的10进制数）获知我选取的一个密文：

```
ad+N7hJlpsVKRgzQU6Xawc5bXy5a8oLuXC3z/U8jDkyxOMgXcEdxdR6s+oyBzbN79yrTj4VhP/
R1TERhCzVpCL8qMjJh841IP2QJ6yFGWfLMLaFb266xyyUGXmNYzERpvONtCnAZABLzWzaSK6kAttDtC6lzKqOgM6apWsivKXk=
```

解码出来的字节的10进制数为：

```
115 116 117 105 100 61 49 49 49 49 49 49 49 49 49 49 124 110 97 109 101 61 228 184
128 228 186 140 228 184 137 229 155 155 228 186 148 229 133 173 228 184 131 229
133 171 57 48 124 102 108 97 103 61 70 97 108 115 101 124 99 111 100 101 61 110 49
103 113 112 54 57 113 112 112 50 53 106 105 48 104 124 116 105 109 101 115 116 97
109 112 61 49 54 50 49 50 55 54 52 52 48 10 10 10 10 10 10 10 10 10 10
```

去掉padding后转换为**UTF-8**即：

```
stuid=1111111111|name=一二三四五六七八90|flag=False|code=n1gqp69qpp25ji0h|timestamp=1621276440
```

然后设定新的明文为（注意`name`字段多了一个`0`，因为`True`比`False`少一位，要保持其它字段尽可能对齐）：

```
stuid=1111111111|name=一二三四五六七八900|flag=True|code=n1gqp69qpp25ji0h|timestamp=1621276440
```

使用`./8/get-new-cbc-cipher.fish`利用之前获得的`allIntermediates`（对应上面*GitHub*链接中的`Pn*`）进行`xor`操作即可获取新的**CBC**密文。由于只改了`flag`临近的字段，所以只有`name`在解密时会变（注意一个汉字占3字节，我设置名字时特意根据16字节长度的block来调整使得解密时只有`name`字段受到影响），基本不会影响解密出来的内容的合法性。

第二个**ECB**需要用**chosen plaintext attack**，我们已经知道明文的格式了，想要获得解密后形状如下明文的密文：

`stuid=1111111111|name=一二<#1F60B>|flag=True|code=abcdabcdabcdabcd|timestamp=??????????`

其中<#1F60B>为如下4字节长度的表情（本**PDF**没有相关字体所以显示不出来）：

图 20.

然后分别设置姓名为“一二<#1F60B>|flag=True|code=”、“一二<#1F60B>abcdabcdabcdabcd”、“一二三”（学号都是`1111111111`）（在此处，使用4字节长度的表情可以方便16字节长度的block的调整。服务器端限制了名字长度不能超过19个字符，但一个字符可以有多个字节。值得指出的是，浏览器中直接提交的话，一个4字节表情会被当作2个字符，导致检查到超出19个字符提交失败；但是如果绕过客户端*JavaScript*检查用`curl`等工具直接提交，服务器端还是只会把4字节表情当作一个字符的），这样解密的3个明文的分别第1-48、33-48、65-96字节的内容将是“`stuid=1111111111|name=一二<#1F60B>|flag=True|code=`”、“`abcdabcdabcdabcd`”、“`|timestamp=??????????`”（最后一个要加上padding）。.

把得到的3个密文的分别第1-48、33-48、65-96字节拼接在一起即可得到新的密文。相关代码见`./8/crack-ecb.fish`。

9 皮浪的解码器

使用*Ghidra*反编译二进制代码，将结果中的变量名和式子调整成更易理解的变量名和式子，得到`./9/decompiled.cpp`代码。

注意到在函数`b64decode`中，若将`code`（即`main`函数中的`enc`）设置为长度为935的**base64**编码字符数组（不包括最后的`'\0'`）（此时仍然能通过`!(max_len <= (code_len >> 2) * 3)`的检查），那么解码出来的`result`字符数组的长度将达到701，然后出错从`b64decode`退回`main`（因为没有添加“=”padding到4的整数倍），这样`dec`多出的1个字节将会覆盖`declen`的最小位的字节，使得`declen`仍然可以通过`!(declen == 700)`的长度检查。如果将这个字节设置为最大的`0xff`，则此时`declen == 0x2ff == 767`，之后打印`dec`时将足以把35字节的的`flag`的所有内容打印出来了。701个`0xff`的**base64**编码去掉最后padding的“=”后是935个“/”，故向服务器发送935个“/”。

这是我第一次参加**CTF**比赛，自然也是第一次做**pwn**类型的题目。我在做涉及二进制代码的题目时在网上搜索了解到了`pwntools`这个**CTF**中常用的*Python*软件包，所以包括本题在内的完成得比较靠后的一些题目，我使用了这个软件包来写脚本（而完成得比较靠前的题目，我则是笨拙地用*fish* **shell**、*JavaScript*、*Python*（没有`import pwn`的情况下）等来写的脚本）。本题代码见`./9/script.py`。

另外值得指出的一点是，我在网上搜索时还了解到了`checksec`这个软件，可以用来检查*C*/*C++*的二进制代码开启了哪些安全措施。例如本题开启了以下安全措施：

图 21.

其中**Partial RELRO**就保证了**.bss**、**.data**区出现在**.got**区之后，其中的全局、静态变量就算溢出也无法覆盖掉**GOT**的内容。而本题中的`enc`、`enclen`、`dec`、`declen`、`flag`全部出现在**.bss**区中，故自然就不能考虑对**GOT**的溢出覆盖攻击了，而应该考虑这些全局、静态变量之间相互的溢出和被覆盖。

这次比赛中有三道**pwn**类题目，而这三道题便分别考察了对静态区（“皮浪的解码器”）、栈（“庄子的回文”）和堆（“←签退→”）上漏洞的利用和攻击。

10 弗拉梅尔的宝石商店

本题我是一血诶>.<。

本题前期给我的最大阻碍是，我电脑上的*Python*的版本是`3.9.5`，而运行`decompyle3`来反编译时就会抛出`KeyError: '3.9.5'`.....于是我不得已下载并且研究了一会儿`pyenv`（因为我平时基本不用*Python*，所以还没有研究过这东西；*NodeJS*倒是用得比较多，所以已经安装过`nvm`），然后用它又安装了一个`3.8.9`版本的*Python*，然后终于可以反编译了.....（事实上，我最开始是想用`uncompyle6`的，因为可以直接用`pip`安装，不需要克隆*GitHub*仓库；然后我发现它需要小于`3.7`版本的*Python*，所以我其实最先用`pyenv`安装了`3.6.13`版本的*Python*；然而，我发现`uncompyle6`反编译不了提供的字节码，用`file`命令一看是`3.8`版本的*Python*字节码.....于是不得已用了`decompyle3`）。反编译的代码见`./10/service.cpython-38.py`。

研究好一阵，没找到代码本身的漏洞，还尝试挨个购买`jade`、`onyx`想看看它们的`desc`是否提供了什么线索。然后突然意识到只要`token`相同，那么用户进行`trade`时打开的文件都是同一个文件：

```
filename = os.path.join('/tmp', token[:5] + token[-5:] + '.txt')
f = open(filename, 'w')
```

这时候就很容易解决问题了：在一个连接里卖0个`citrine`，自然能通过`check_transaction`；通过之后，先不输入`y`确定交易，而是另外打开一个连接，卖10000个`egg`，这个连接虽然会失败，但是覆盖了`filename`文件的内容，这时候前一个连接再确认交易，就空手套白狼卖出10000个`egg`，就有钱买`flag`了。但是`flag`不能直接买，还需要用相同的方式再操作一次，就能买到了。flag就在`flag`的`desc`中。

11 未来的机器

这道题我也是一血=皿=但其实我根本没有学过汇编诶。

*Python*代码本身还是容易理解的，其实就是实现了一个类似于汇编语言的低级语言的解释器。可以通过在`run`函数中插入相关代码实现一些debug的功能。该代码我修改的最后版本见`./11/runner.py`。

然后我就开始疯狂等价性简化汇编代码（比如说含有`114514`的行都被我无情地去掉了=A=）。最后简化的版本见`./11/asm.txt`。

但是我并没有简化全部的汇编代码。开始时我是从前面开始简化的，后来意识到前面有些代码可能后面根本用不上，于是就改成从最后开简化，然后看后面的代码访问了前面的哪些变量，然后再到前面相应代码块去做简化。另外很重要的一点是先把记录了重要`Memory`的index的全部变量都先替换成对应数字。

最后发现，用户输入后：

1. 输入的内容`inv`先被编码放到`MEMORY[4982: 4982 + len(key)]`中（`inv`的长度必须与`key`相同才能通过检验，所以这里的`len(key)`其实就是`len(inv)`）（真实情况其实是`MEMORY[4982: 4982 + len(key) * 4]`，但由于*Python*的整数没有大小限制，不会溢出，所以`MEMORY`中连续每4个单元格只有一个用来存储数据，在这里就直接简化为`MEMORY[4982: 4982 + len(key)]`，下同；下面的`MEMORY[1152 + index]`等也应理解为`MEMORY[1152 + index * 4]`）；

2. 若`MEMORY[4982: 4982 + len(key)]`有小于$(0+32)$或者大于等于$(96+32)$的，程序便会出错退出；

3. `MEMORY[4982: 4982 + len(key)]`中的内容在后面会被用来调整`MEMORY[1152: 1152 + len(key)]`的值，具体算法是（其中`MEMORY[2336: 2336 + len(key)]`为`list(range(len(key)))`的一个排列）：

   ```
   MEMORY[1152 + index] = \
       (MEMORY[1952 + (MEMORY[4982 + index] - 32)] + index) % 96 + 32
   ```

4. `MEMORY[1152: 1152 + len(key)]`中的内容在后面会被用来调整`MEMORY[1552, 1552 + len(key)]`的值，具体算法是（其中`MEMORY[1952: 1952 + 96]`为`list(range(96))`的一个排列）：

```
MEMORY[1552 + MEMORY[2336 + index]] = MEMORY[1152 + index]
```

5. 最后`MEMORY[1552: 1552 + len(key)]`被用来和`key`进行比较，若一样的话就成功；若不一样则失败。

将整个过程逆转过来，即可求出flag，代码见`./11/script.py`。需要指出的一点是，解密算法需要使用到`MEMORY[1952: 1952 + 96]`和`MEMORY[2336: 2336 + len(key)]`的值，但我们不需要从开头一点一点看这些值是怎么算出的，而只需要在程序中使用这些值的行处插入断点，到时候直接打印出这些值即可。

12 庄子的回文

使用*Ghidra*反编译二进制代码，将结果中的变量名和式子调整成更易理解的变量名和式子，得到`./12/decompiled.cpp`代码。

使用`checksec`检查，结果如下：

```
[I] say@Arch-on-Gram ~/C/L/p/zhuangzi> checksec --file=pwn
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols       FORTIFY Fortified      Fortifiable     FILE
Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   No RUNPATH   71) Symbols     No    0                1               pwn
```

图 22.

可以发现没有开启**PIE**，该二进制代码中的`main`函数等的地址都是已知的，所以可以使用**return-oriented programming**，利用读入字符串时的**stack buffer overflow**来覆盖掉返回指针，打印出`libc`的基址并返回`main`函数，然后再次溢出覆盖，执行`system("/bin/sh")`获取**shell**。我基本参考的an intro to ret2libc & pwntools (64bit)这篇文章的内容（也同时学习到了使用`pwntools`），代码见`./12/script.py`（注意，我使用的*Linux*发行版是*Arch*，我写的本地运行代码能在我的电脑上正常运行，但并不能在*Ubuntu*操作系统的电脑上运行；为了适配*Ubuntu*系统的服务器，我写的针对服务器的代码增加了栈对齐部分）。

13 无法预料的问答

这道题，其实就是出题者给**emoji**（**emoji**的复数形式既可以是“**emojis**”，也可以就是“**emoji**”）添加了一个严格的全序关系，然后每次进行选择时，从所给几个**emoji**中选择在该严格全序关系中最大（也可以说是最小）的那个emoji即可。因此要进行足够多次的尝试，获得较多的**emoji**，就可以尽可能详尽地知道这个严格全序关系，并做到连续20次答对。

我实现的代码见`./13/script.js`。基本算法思路是构建一个映射，将每个**emoji**映射为由和它出现过在同一道题并且根据服务器反馈而确定比该**emoji**更大的bigger **emoji**（例如当时选了这个bigger **emoji**，服务器反馈说选择正确；或者当时有2个**emoji**确定比其它的**emoji**更大，然后选择了其中一个**emoji**提交给服务器，服务器说选择错误，而另一个**emoji**则是bigger **emoji**）所组成的列表，然后其实这些列表中的bigger **emoji**也能再找到比它们更大的bigger-bigger **emoji**列表。如此递归下去，我们就能知道根据已知的信息，哪些**emoji**确定比某个**emoji**更大。对于每一道题，如果我们能据此确定出一个最大的**emoji**，那么就提交它；否则就随便提交一个答案，然后根据服务器的反馈来更新映射。提供的文件`./13/map.json`就是这么一个映射。

14 安全的密钥交换

这道题想要说明的其实就是如果不存在最初的证书（或者说“信任”）的话，无论用Diffie–Hellman密钥交换算法，还是用**RSA**加密算法，都没办法防范中间人攻击。在**TLS**中，解决这个问题的办法是提前给设备安装好根证书；在**OpenPGP**中，靠的则是先通过线下面对面交换等安全方式交换公钥，然后再逐渐通过互相签名构建出一个信任网络（**web of trust**）。

具体到这道题上面，先分别收取Alice和Bob互相发给对方的消息，然后随便选择一个自己的密钥，比如选择0，使用Diffie–Hellman密钥交换算法进行处理然后将新的消息发送给Alice和Bob。接下来就能够收取Alice和Bob发送给自己的证书了。这是第一次连接服务器所要干的事情，代码见`./14/session-1.ipynb`。

第二次连接服务器时，前面做的事情都一样（将自己的密钥选择为0），但是在收到Alice发过来的证书时，把Bob的证书发给Alice，就能让Alice相信发信者是Bob（其实这件事情在第一次连接服务器时就可以做了，但是为了给每次连接服务器一个明确的任务，我将其安排在了第二次）。因为我们将自己的密钥选择为了0，所以Alice那么的`key`计算出来的结果是1，就会把第一个flag发送给我们。代码见`./14/session-2.ipynb`。

第三次连接服务器时，做的事情和第二次基本一样，只是在选择自己的密钥时暴力枚举使得用Diffie–Hellman密钥交换算法得出的结果大于等于2^{512}即可。代码见`./14/session-3.ipynb`。

15 计算概论B

猜测使用的是Huffman编码。于是按照各个16进制数出现的次数作为权重，在网上找了一个在线Huffman编码器获得其中一种Huffman编码，计算编码后的二进制序列总长度，发现的确和提供的二进制序列总长度一致。然后就对网上已有的*Python*写的Huffman编码代码进行调整，使得其可以求出所有可能的Huffman编码（应该是有$2^{16}=65536$种），然后暴力对每一种Huffman编码进行验证，验证方式是看解码后各个16进制数出现的次数是否与题目中提供的一致（此处其实也可以检验`binascii.hexlify(b'flag{'[: : -1])`是否出现在了解码结果中）。最后发现2种Huffman编码满足条件，但只有其中一种解码后的结果包含`binascii.hexlify(b'flag{'[: : -1])`，由此得到flag。本题相关代码见`./15/game.ipynb`。

16 巴别压缩包

最开始我还傻傻研究了一下**ZIP**格式，以及题目提供的压缩包中的相关信息（见`./16/hex.odt`）。后来没研究出结果，就上网查关于这种递归压缩包的信息（还真找到了“从 42KB 文件中解压出 4.5PB”所指的**zip bombs**的介绍），并且发现有专门的关于`quine.zip`的介绍和讨论，例如https://research.swtch.com/zip和https://matthewbarber.io/gzip-quine/。最终发现还是需要暴力破解……于是参考已有代码进行调整，得到了代码`./16/find-crc32.py`。不知为何，虽然该代码使用了多进程，但在我电脑上跑得非常非常慢（可能正是因为用了多进程=A=），而且还很影响我电脑性能、带来严重的发热，于是我决定同时在我的*Android*手机上运行（使用*Termux*中的*Python*）。但手机上不支持*Python*的`multiprocessing`库，于是我又改写了一个手机的单进程适配版本，见`./16/find-crc32-android.py`（这个代码在手机上运行的速度似乎比那个多进程代码在笔记本电脑上运行的速度更快一些……）。事实上，我还有几个**VPS**服务器，所以其实还可以同时在这些服务器上面跑，但因为时间并不是非常急迫，所以我没有这么做（当然，写一个使用自己电脑的**GPU**的并行计算程序，应该是最节省时间的）。最后就把结果给暴力算出来了。

17 ←签退→

作为一个此前从来没有做过**pwn**题的人，我花了相当多的时间来研究这道题。最后发现，这道题的思路其实很常规。

使用*Ghidra*反编译二进制代码，将结果中的变量名和式子调整成更易理解的变量名和式子，得到`./17/decompiled.cpp`代码。

接着还是使用`checksec`检查，结果如下（可以看到开了**Full RELRO**和**PIE**，所以也没办法像“庄子的回文”那样或者靠覆盖**GOT**来攻击）：

```
[I] say@Arch-on-Gram ~/C/L/p/sign-out> checksec --file=pwn
RELRO           STACK CANARY      NX           PIE           RPATH      RUNPATH      Symbols       FORTIFY   Fortified      Fortifiable    FILE
Full RELRO      No canary found   NX enabled   PIE enabled   No RPATH   No RUNPATH   72) Symbols     No      0              3              pwn
```

图 23.

代码中有再明显不过的**use-after-free**的漏洞，而且不但读了一次，还写了一次（我们之后就把这个指针记为`name`，与我调整过后的反编译代码相一致）。

读的这一次就泄漏了`libc`的基址，因为`name`被`free`过后会进入**unsorted bin**，其前向指针`name->fd`和后向指针`name->bk`都被填充进了指向`main_arena + 0x60`位置的指针，其中前向指针`name->fd`会在`printf("Now␣your␣name␣is:%s,␣please␣input␣your␣new␣name!\n", name);`这句中被打印出来（由于`name->fd`必定是`0x00007fXXXXXXXXXX`的形式，其末尾的`'\0'`便阻止了`name->bk`被打印出来，不过它俩是一样的）。而`main_arena`与`libc`的基址之间的偏移量是一定的（事实上，`main_arena`就在`__malloc_hook`的`0x10`之后，所以用`pwntools`可以有`libc.address = mainArenaPointer - 0x10 - libc.symbols["__malloc_hook"]`。

而写的这一次，则能让我们进行**unsorted bin attack**。由于大小刚好一样，所以最后一次`malloc`时得到的指针指向的地址恰好是`name`指向的地址。如果我们提前把某个地址（准确地说是该地址`-0x10`）写进`name->bk`的位置，那么在之后这次`malloc`时就能将`main_arena+0x60`写进该地址中，实现往任意地址写一个大数（`0x7fXXXXXXXXX`）的效果。由于我们已经有了`libc`的基址了，所以我们可以往`global_max_fast`处覆盖这个巨大无比的数，这样最后`free`的两个boxes的内存都将因大小小于`global_max_fast`而进入**fastbin**。遗憾的是，我的电脑上的*GNU libc*的版本是`2.33`，在这个版本中，`malloc`增加了新的检查，如果直接往`name->bk`处写入`global_max_fast-0x10`，而`malloc`又发现`global_max_fast`处没有相应指针的话，就会直接打印错误信息“`malloc(): unsorted double linked list corrupted`”并终止程序。我最终找到了一个绕过这个检查的方法，那就是先用`pwntools`打开程序，但是在写入`name->bk`前暂停，然后将`gdb`给attach到这个进程上去（`pwntools`可以直接开启`gdb`，但不知为何在我的电脑上`gdbserver`被打开过后就一直不会动，所以只能用这个更原始的方法），然后在`_int_malloc+1356`处设置断点（`break *_int_malloc+1356`），在此处直接跳跃到`_int_malloc+1371`（`jump *_int_malloc+1371`），就把检查给绕过去了。

之后，注意到`__printf_arginfo_table`、`__printf_function_table`与`main_arena`之间的距离的两倍减去`0x10`均恰好在两个boxes可以选择的大小之间，因此我们就完全可以选择适当的boxes的大小，达到往这两个地方写入东西的目的，并且在最后一次拥有编辑box的机会时提供**one gadget**的地址（可以用`https://github.com/david942j/one_gadget`找到），使得`free`时把该地址给写入。最后使用`printf`时我们便能获得**shell**了。事实上，这道题基本就是**House of Husk**的翻版。

再一次，非常的遗憾，由于我电脑上的*GNU libc*的版本和服务器上的非常不一样，很多偏移量的结果也就不一样，而且这也给我debug造成了非常严重的问题，使得我没办法做完这道题……我在网上找到一个叫做`pwn_debug`的软件，号称可以同时使用多个版本的*GNU libc*。然而我却看到它的脚本中在运行`apt-get`……（BTW, I use *Arch*.）懒得去调它的脚本，也懒得专门装个虚拟机。希望今后主办方能提供具有相关环境的机房电脑，以便使用不同环境的参赛选手也能顺利debug（或者有其它简单的方式可以更换使用的*GNU libc*也请告知参赛选手，我试了`LD_PRELOAD`、`LD_LIBRARY_PATH`环境变量以及用`patchelf`似乎都不太行，当然也可能是我操作有误）。

我把我半成品的代码放在`./17/script.py`，从里面可以看出我进行过相当多的尝试（有的尝试还很幼稚）。

为了做这道题，我从0开始学习对堆的漏洞的利用，看了巨多资料，在下面列出其中一部分（还有更多的标签页已经关掉了，懒得去历史浏览记录里面去找，感兴趣的同学可以自己用*Google*搜索相关资料），以供看不懂本文中使用的术语的读者阅读：

1. **House of Husk**
2. picoCTF 2019: Heap Exploitation Challenges (Glibc 2.23, 2.27, 2.29)
3. **Unsorted Bin Attack**
4. Educational Heap Exploitation
5. **Use-After-Free**
6. 0ctf 2016 - Zerostorage
7. Heap Master Review

8. **House Of Corrosion**
9. Heap overflow
10. HITCON CTF Qual 2016 - House of Orange Write up
11. **house of orange** in glibc 2.24