虚拟机保护逆向入门

原创

[合天网安实验室](#) 于 2018-06-08 18:00:00 发布 232 收藏 1
版权声明：本文为博主原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接和本声明。
本文链接：https://blog.csdn.net/qq_38154820/article/details/106329759
版权

点击上方蓝字 即刻关注我们

本文原创作者：b1ngo

原创投稿详情：重金悬赏 | 合天原创投稿等你

前言

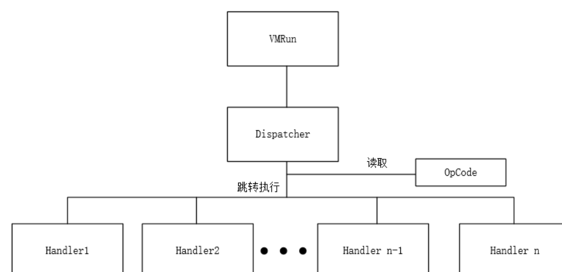
软件防逆向工程与逆向工程相伴发展，早期的有花指令，反调试技术，代码混淆与加密，加压缩壳或者加密壳等等保护手段，这些技目前已经有了较好的解决方案，自动化的分析方法也比较成熟。目前比较前沿的软件保护技术是虚拟机保护（Virtual-Machine-Protect），当然这种虚拟化的思想也广泛用于软件开发等其他领域。

最近在打一些CTF比赛的时候，感觉虚拟机保护的逆向已经成了Reverse题目的一个新趋势，甚至现在的CTF中，稍微简单的虚拟机题目都快成入门题了。当然这里的虚拟机往往不是商业的vmp之类的保护软件，而是出题人实现的小型虚拟机，题目本身一般也没有实现某些复杂的功能。本篇文章简单的介绍一下虚拟机保护的基本原理，并记录一些最近的几个题目。受限于笔者能力水平，有错误望指正交流。

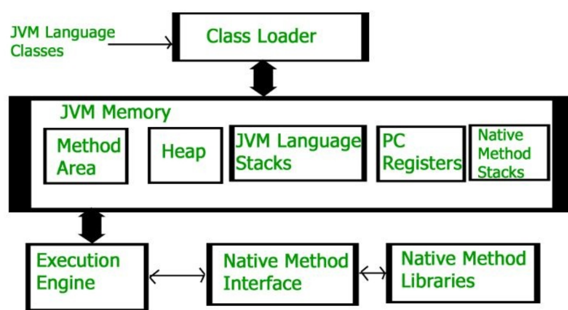
虚拟机保护原理

基本原理

这里的虚拟机当然并不是指VMWare或者VirtualBox之类的虚拟机，而是指的意思是一种解释执行系统或者模拟器（Emulator）。所以虚拟机保护技术，是将程序可执行代码转化为自定义的中间操作码（OperationCode，如果操作码是一个字节，一般可以称为Bytecode），用以保护源程序不被逆向和篡改，opcode通过emulator解释执行，实现程序原来的功能。在这种情况下，如果要逆向程序，就需要对整个emulator结构进行逆向，理解程序功能，还需要结合opcode进行分析，整个程序逆向工程将会十分繁琐。这是一个一般虚拟机结构：



这种虚拟化的思想，广泛用于计算机科学其他领域。从某种程度上来说，解释执行的脚本语言都需要有一个虚拟机，例如LuaVM，PythonInterpreter。静态语言类似Java通过JVM实现了平台无关性，每个安装JVM的机器都可以执行Java程序。这些虚拟机可以提供一种平台无关的编程环境，将源程序翻译为平台无关的中间码，然后翻译执行，这是JVM虚拟机的基本架构（图片来自geeksforgeeks）：



从这个解释中，我们可以看到虚拟机保护有其缺点，就是程序运行速度会受到影响。在商用的一些虚拟机保护软件中，可以提供SDK，在编程的时候可以直接添加标记，只保护关键算法。

分析方法

从这里开始，本文所指的虚拟机都是私有实现的小型虚拟机，并不是指成熟的商用虚拟软件，例如VMProtect、Themida等，这些保护软件的逆向工作研究工作很多，不过比较完善和系统的解决方案还没有出现。

在目前的CTF比赛中，虚拟机题目常常有两种考法：

·给可执行程序和opcode，逆向emulator，结合opcode文件，推出flag

·只给可执行程序，逆向emulator，构造opcode，读取flag

拿到一个虚拟机之后，一般有以下几个逆向过程：
·分析虚拟机入口，搞清虚拟机的输入，或者opcode位置

·理清虚拟机结构，包括Dispatcher和各个Handler

·逆向各个Handler，分析opcode的意义

在实际调试中，个人觉得最为关键的是要搞清虚拟机的执行流。调试过程中，在汇编层面调试当然是最基本最直接的方法，但是由于虚拟机Handler可能比较多，调试十分繁琐。若虚拟机内部没有很复杂的代码混淆，可以考虑使用IDA进行源码级调试，这对于快速整理emulator意义很有帮助。再进一步，可以结合IDA反编译伪代码，加上一些宏定义，加入输出，重新编译，可以十分快速的逆向整个emulator执行过程。

目前比赛中，虚拟机题目特点是核心算法不是很复杂，虚拟机本身没有反调试和代码加密混淆的加入。当然，随着整体CTF水平的不断进步，未来虚拟机逆向难度只会越来越高。

题目示例

在这一部分，我选择了最近3个VM类题目，题目可以在这里下载（https://github.com/b1ngoo/VMCTF），限于篇幅（太长估计也看的不舒服2333），就只写3个题目了。

DDCTF 2018黑盒破解

这个题目可以属于给二进制程序，构造opcode，令程序输出Binggo字样，

然后在服务器上输入该opcode，可以读取flag。

file命令看一下：64位的ELF文件

```
└─[$]> file ReverseMe.elf
ReverseMe.elf: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /l
b7b4a79471c215fafd179, stripped
┌─[b1ngo@b1ngo-VirtualBox] - [~/Reverse] - [五 4月 27, 08:31]
```

IDA中看下逻辑，main函数十分清楚。然后跑一下，题目要求是这样的：首先输入password（就是给的txt文件中的数字），然后要求输入passcode，令程序输出Binggo。本地测试通过之后，然后将该passcode提交到服务器上，就会得到flag。

main函数最关键的判断在这里：

```
if ( memcmp("exit", (const void *)(v4 + 16), 4uLL) )
{
  sub_401A48(v4);
  if ( byte_603F00 )
    printf("Success!\nYour flag is %s\n", &ptr);
  else
    puts("Failed!");
}
```

查找引用，找到sub_40133D函数，如果对这个函数再次查找应用，又找到了14个函数：

```
000603840 off_603840      dq offset sub_400DC1    ; DATA XREF: sub_4016BD+239↑r
000603848                 dq offset sub_400E7A
000603850                 dq offset sub_400F3A
000603858                 dq offset sub_401064
000603860                 dq offset sub_4011C9
000603868                 dq offset sub_40133D
000603870                 dq offset sub_4012F3
000603878                 dq offset sub_4014B9
000603880                 dq offset sub_400CF1
000603888                 dq offset sub_400E22
000603890                 dq offset sub_400FA0
000603898                 dq offset sub_400EEA
0006038A0                 dq offset sub_40121D
0006038A8                 dq offset sub_4010F5
0006038B0                 db    0
```

但是其实我们还是没有找到调用sub_40133D函数的地方，这里我就陷入了僵局。这时随意翻看main函数，在关键判断之前还调用了一个sub_401A48函数，并且该函数的参数就是我们输入的passcode，进去看下，如果反编译看伪代码的话，其实是十分不友好的，但是如果看CFG图的话，还是可以看出来这是一个虚拟机的Dispatcher位置，在这个基本块里面通过callrax跳入到每个Handler里面：

```
jnz     short loc_401B6D

mov     rax, [rbp+var_28]
mov     edx, [rbp+var_18]
movsxd  rdx, edx
add     rdx, 48h
mov     eax, [rax+rdx*4+8]
mov     [rbp+var_14], eax
mov     rax, [rbp+var_28]
mov     edx, [rbp+var_14]
movsxd  rdx, edx
add     rdx, 54h
mov     rax, [rax+rdx*8+8]
mov     rdx, rax
mov     rax, [rbp+var_28]
mov     [rax+2A0h], rdx
mov     rax, [rbp+var_28]
mov     rax, [rax+2A0h]
mov     rdx, [rbp+var_28]
mov     rdi, rdx
call    rax
```

接下来通过逆向这部分代码找到可以输入的passcode，并且可以找到对应关系：

“””

sub_400DC1——> $ func1

sub_400E7A——> 8 func2

sub_400F3A——> C func3

sub_401064——> t func4

sub_4011C9——> 0 func5

sub_40133D——> E func6

sub_4012F3——> u func7

sub_4014B9——> # func8

sub_400CF1——> ; func9

“””

然后给出相对于a1偏移存储的内容：

“””

(a1+ 288)：指向字符串的索引位置

(a1+665)：临时存放变量位置

(a1+664)：输入的每个值位置

“””

然后通过再次动调，我们可以了解到我们这部分的任务：通过输入字节码，调用不同函数（func1~func9），将长度为20的字符串PaF0!&Prv}H{ojDQ#7v=转为长度为6的字符串Binggo。然后在func6函数中，做检测，如果是Binggo，就满足条件。

接下来的工作就是一个个看这几个func，通过伪代码大致可以理解一下：

“””

func1:

*(a1+665)= str[a1+288]

func2:

str[a1+288]= *(a1 + 655)

func3:

*(a1+665)= *(a1+665) + *(a1+664) - 33

func4:

(_BYTE)(a1 + 665) = *(_BYTE *)(a1 + 665) - *(_BYTE *)(a1 + 664) + 33;

func5:

++*(a1+ 288)

func6:

put(str,20)

func7:

--*(a1+ 288)

func8:

str[a1+288]= input[*(a1+288) + *(a1+664) - 48] - 49

func9:

for(i = 0; *(a1 + 664) > i; ++i)

++*(a1 + 288)

str[a1+288]= input[*(a1+288) + *(a1+664) - 48] - 49

“””

最后慢慢构造出满足条件的passcode，还有别忘了构造完成后需要将字符串索引通过func7移到开始位置，然后调用puts函数输出。构造方法有很多种，我是从头往后构造，看了其他人的writeup，也有师傅是从后往头构造。这是我的passcode：

“””

$t/80$C)80$CI80$CX80$Cg80$Cj80#J1uuuuuuuEs

“””

本地验证一下通过：

```
Please input your password!
1234567890
--------------------[Welcome To ReverseMe!]--------------------


Please Input Your Passcode,If You See print the "Binggo" string,Congratulations,You Win. Good luck!
$t/80$C)80$CI80$CX80$Cg80$Cj80#J1uuuuuuuEs
Binggo
Success!
Your flag is DDCTF{ThisIs_AnExample}
```

远程也是可以通过的：

```
└─[$]> nc 116.85.48.236 5000
Please input your password!
48ee204317
--------------------[Welcome To ReverseMe!]--------------------


Please Input Your Passcode,If You See print the "Binggo" string,Congratulations,You Win. Good luck!
$t/80$C)80$CI80$CX80$Cg80$Cj80#J1uuuuuuuEs
Binggo
Success!
Your flag is DDCTF{ca88abb361955748}
```

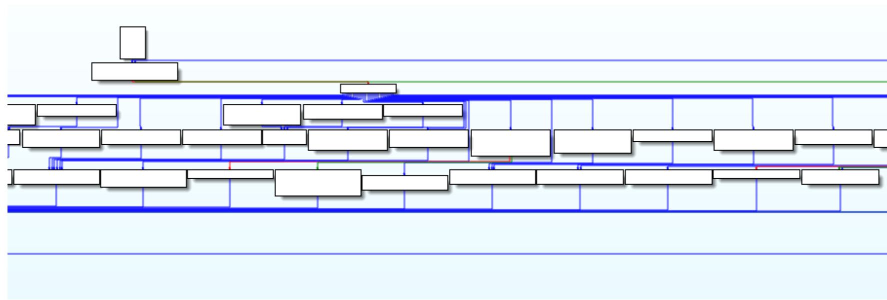
RCTF 2018 Simple vm

这个题是今年RCTF2018的一个入门题吧，感谢这位师傅的出题和分享，学

到了很多。题如其名，这是一个虚拟机程序，题目本身给了一个ELF64位的可执行程序，给了一个bin文件，这是opcode文件。

vm_run在sub_400896函数，进去之后发现没有代码加密或者混淆，直接可

以反编译和生成CFG图：



从结构可以看出虚拟机一般的switch/case结构，所以loc_4008A2可以理解为vm_dispatcher，各个case为vm_handler。赛后做这个题目，我们就用最傻的方法，一步步调试这个vm，结合opcode，理解各个handler。在IDA的反编译结果中，出现了两个比较关键的全局变量：dword_6010A4和c。我们可以将之认为是用于临时存放opcode指令和代码的变量。

给出p.bin文件的解析：

“””

01 # 读取addr赋值给指针

30000000 # 带读取的addr

15 # 读取字符串所在addr赋值给vm.c

00010000 # Inputflag字符串长度地址，后面就是字符串

#dword_6010A4是字符串的ASCII，c是字符串所在的地址

这部分完成了打印字符串

0E # c++

12 # dword_6010A4 = p[c]

0B # putchar(dword_6010A4);

0C # 循环

00010000 # 循环次数所在地址（也就是字符串的长度）

35000000 # 每次循环开始的指针

这部分完成了循环

15 c = READ_INT();

10010000 # 要读入的字符串长度地址

0E # c

0A # dword_6010A4 = getchar();

66 # nop

16 # p[c] = dword_6010A4存放到0x111位置处

0C # 循环

10010000 # 循环的次数

47000000 # 每次循环开始的指针

03 #取0x140偏移处的内容0x20赋值给dword_6010A4

40010000

10 # c = dword_6010A4;

11 # dword_6010A4 += val;

F1000000 # val

13 # dword_6010A4 =p[dword_6010A4]; 到这个位置，就将输入的ASCII赋值到dword_6010A4中了

04 # p[dst] = dword_6010A4;

43010000 # dst 0x143

#开始对输入做变化，0x141~0x144可以理解为变化临时存放为位置

08 # dword_6010A4 =~(dword_6010A4 & c)

04 # 将dword_6010A4存放到后面的地址上

41010000 # 0x141地址

10 # c = dword_6010A4将一次变化后的值给c，进行迭代变化

03 # 取0x140处内容给c

40010000 # 0x20

08

04

42010000 # 将第二次变化内容复制到0x142地址处

03 # 取第一次变化后的值给dword_6010A4

41010000 # 存放第一次变化后的值

03 # 取每个输入值

43010000 # 每个输入值存放位置

08 #

10 # c = dword_6010A4

03 # 取第二次变化给dword_6010A4

42010000

08 # 最后一次变化

0444010000 #

03 # dword_6010A4赋值

40010000

11 # dword_6010A4 add

F1000000 # add value

10 # c = dword_6010A4

03 #将一个地址处内容赋值给dword_6010A4,这里是将变化后的输入赋值给dword_6010A4

44010000 # addr,存放输入变化后的值

16 # p[c] = dword_6010A4

05 # 设置c为0x20

40010000

0E # c++

06 # 将c设置到0x140位置

40010000

0C # loop

45010000 # loop 次数

55000000

ch(49,32) = -33

ch(32,ch(49,32)) = -1

ch(49,ch(49,32)) = -18

ch(ch(32,ch(49,32)) ,ch(49,ch(49,32))) = 17

====> xor(49,32) = 17

====> xor(input[i],i+0x20)

03 #取地址0x146位置的0x1f赋值给dword_6010A4

46010000 # 地址参数

11 # add

05000000

13 # dword_6010A4 =p[dword_6010A4]

10 # c = dword_6010A4

03 # dword_6010A4 = p[src]

46010000 #

11 # add dword_6010A4 += val

11010000

13 # dword_6010A4 =p[dword_6010A4]

17 # dword_6010A4 -= c;

18 # OP_JNE

60010000

0C # 循环

46010000

B6000000

“””

这就是这个程序的执行流，调试过程十分麻烦，vm实在是太磨人了。理解

上述过程，其实就是对输入做异或，并与p.bin文件部分内容进行比较的过程，然后写出逆向算法，解出flag：

“””

-*- coding:utf-8 -*-

a ="101843141547401710 1D4B121F4948185354015751530556 5A08585F0A0C5809"

a = a.decode("hex")

a = [ord(i) for i in a]

print a

f = ""

for i in range(len(a)):

f += chr(a[i]^(i+32))

print f

“””

其实这种方法算是最麻烦的，该方法是从汇编层面调试二进制程序，我还看

到更为简单的一种方法是通过IDA反编译结果（https://expend20.github.io/2018/05/24/RCTF-simple-vm.html），稍微改改之后，重新编译，然后就可以进行源码级调试。这里的修改基本不用动IDA的反编译结果，就是需要加些IDA的宏定义，在每个case后面都使用printf输出当前指针位置，就可以自动的分析出程序的执行流，十分方便。

从赛后复盘的角度是可以慢慢调试虚拟机的，搞清每个字节码的意义，然后

完整的理解虚拟机解释执行过程。不过随着虚拟机的复杂，这种方法将会非常费时，应该学习一些自动化的处理方法。

RCTF 2018 magic

又是今年RCTF的一个Re题，题目也出很好，我感觉还挺难的，赛后看了一

些其他师傅的WP，感觉收获很多。这里前面爆破time的第一次检查不再赘述（吾爱有个师傅的处理方法很不错https://www.52pojie.cn/thread-742361-1-1.html），只说说第二个对输入的检查。当第一次对时间检查通过之后，程序执行流来到了函数sub_4023B1位置：

```
__int64 __fastcall sub_4023B1(const char *a1, char *a2)
{
  __int64 v2; // rdx
  __int64 v4; // [rsp+20h] [rbp-30h]
  __int64 v5; // [rsp+28h] [rbp-28h]
  __int64 v6; // [rsp+30h] [rbp-20h]
  __int64 v7; // [rsp+38h] [rbp-18h]
  unsigned int v8; // [rsp+43h] [rbp-Dh]
  char v9; // [rsp+47h] [rbp-9h]
  __int64 *v10; // [rsp+48h] [rbp-8h]

  if ( !dword_4099D0[0] )
    exit(a1);
  v9 = 0;
  v8 = dword_4099D0[0];
  sub_402218(a1, a2, 49, &unk_4052A0, dword_4099D0[0]);
  v4 = 0LL;
  v5 = 0LL;
  v6 = 0LL;
  v7 = 0LL;
  v10 = (&v4 + 4);
  scanf(a1, a2, &v4 + 4, "%26s");
  sub_401F37(a1, a2, 26LL, v10, &v8);
  if ( !vm_run(a1, a2, v2, v10) )                 // 进入虚拟机
    return sub_402218(a1, a2, 6, &unk_4052D1, dword_4099D0[0]);
  sub_401F37(a1, a2, 26LL, v10, &v8);
  sub_401FFB(a1);
  sub_402218(a1, a2, 35, &unk_4052E0, dword_4099D0[0]);
  puts(a1);
  return sub_402218(a1, a2, 35, &unk_4052E0, dword_4099D0[0]);
}
```

scanf之后，对输入进行加密，加密后的数据作为虚拟机输入。这个地方是利用setjmp/longjmp实现的一个虚拟机，虚拟机opcode位于

0x405340处。地址409040是虚拟机用于数据处理的区域，我们可以将之定义为dword型数组。该虚拟机值得主要的是这个地方，调试的过程中需要注意一下，这里是通过异常处理完成了一个Handler：

```
      goto LABEL_43;
    dword_409060 = byte_405340[v10] >> 4;
    dword_409064 = byte_405340[v10] & 0xF;
    if ( !setjmp(Count) )
      byte_405340[v10] = dword_409060 / byte_405340[v10 + 1];// 除0异常
    v10 += 2;
  }
```

异常处理在这个signal回调函数中：

```
void __fastcall __noreturn Func(int a1)
{
  if ( a1 == 8 )
  {
    signal(8, Func);
    *(&dword_409040 + dword_409060) = *(&dword_409040 + dword_409060) == *(&dword_409040 + dword_409064);
    sub_404716(Buf, 0i64);
  }
  exit(1);
}
```

对于虚拟机的逆向，往往都是通过首先逆出来各个Handler，然后结合opcode

搞清楚程序算法，在目前这个阶段，这个算法一般不会很难。如果算法很难，opcode很长，这种情况题目就算题出的很难的。接下来给出opcode的解释：

“””

con[1] = cmp

con[2] = input

AB 03 00 mov con[3],0

AB 04 1A mov con[4],0x1A

AB 00 66 mov con[0],0x66

AA 05 02 mov con[5],con[2] #input

A9 53 con[5] += con[3]

A0 05 mov con[5],*(byte)con[5]

AB 06 CC mov con[6],0xCC

A9 56 con[5] += con[6]

AB 06 FF mov con[6],0xFF

AC 56 con[5] &= con[6]

AE 50 con[5] ^= con[0]

AD 00 not con[0]

AA 06 05 mov con[6],con[5]

AA 05 01 mov con[5],con[1]

A9 53 con[5] += con[3]

A0 05 mov con[5],*(byte)con[5]

AF 56 00 if con[5]==con[6]

算法总结：

key = 0x66

((input[i] + 0xCC) &0xFF) ^key == cmp[i]

~key

“””

直接爆破就可以得到需要输入值：

“””

comp ="89C1EC50973A5759E4E6E442CBD90822AE9D7C07808F1B4504E8"

comp = [ord(i) for i incomp.decode("hex")]

print comp

'''

key = 0x66

((input[i] + 0xCC) &0xFF) ^key == cmp[i]

~key

'''

key = 0x66

f = []

for i in range(len(comp)):

for j in range(256):

if ((j + 0xCC) &0xFF) ^ key == comp[i]:

f.append(j)

key = (~key) & 0xFF

print f

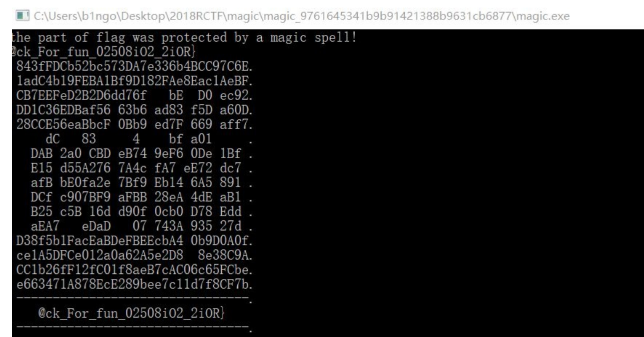
“””

值得注意的是，这个只是vm_run的输入值，需要利用程序中的解密算法解

密后得到输入值，这个可以在vm_run成功后，在地址0x040249D下断点，在内存将要解密的值进行替换，最后解密后如下所示：

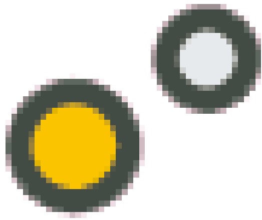
```
十六进制                                         ASCII
40 63 6B 5F 46 6F 72 5F 66 75 6E 5F 30 32 35 30  @ck_For_fun_0250
38 69 4F 32 5F 32 69 4F 52 7D 00 00 08 00 00 A4  8iO2_2iOR}.....¤
E7 2C 32 00 84 FD 60 00 00 00 00 00 C0 14 B9 00  ç,2..ý`.....À.¹.
00 00 00 00 75 32 40 00 00 00 00 00 08 00 00 00  ....u2@.........
```

这个只是其中一部分的flag，将之输入后会得到一个图案，图案为rctf{h:



拼接之后即为flag。

看不过瘾？合天2017年度干货精华请点击《【精华】2017年度合天网安干货集锦》



别忘了投稿哦！

合天公众号开启原创投稿啦！！！

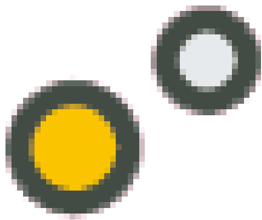
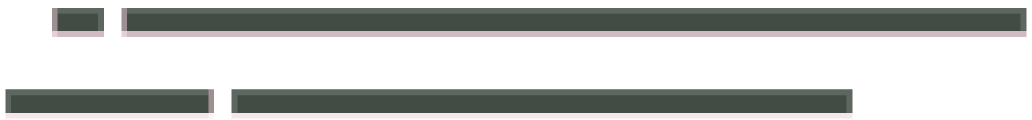
大家有好的技术原创文章。

欢迎投稿至邮箱：edu@heetian.com

合天会根据文章的时效、新颖、文笔、实用等多方面评判给予100元-500元不等的稿费哟。

有才能的你快来投稿吧！

点击了解投稿详情 重金悬赏 | 合天原创投稿等你来！

创作打卡挑战赛
赢取流量/现金/CSDN周边激励大奖