

CTF小白学习笔记(Reverse)-i春秋 NoExec

原创

[Istill...](#) 于 2020-12-10 15:37:43 发布 506 收藏 2

分类专栏: [CTF的writeup](#) 文章标签: [安全](#) [信息安全](#)

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_44370676/article/details/110949986

版权



[CTF的writeup](#) 专栏收录该内容

8 篇文章 0 订阅

订阅专栏

大纲

一.解题过程

预处理

DOS ME头IMAGE_DOS_HEADER

PE文件头

找到入口

分析check函数

字符串按索引奇偶分成2个字符串

将2个字符串base64编码后拼接

将拼接后的字符串DES加密

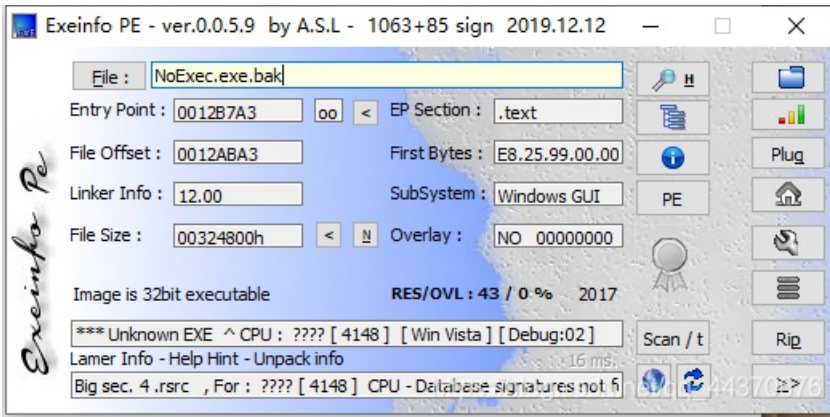
再一次base64

二.解题脚本

三.总结

一.解题过程

这道题涉及到的有windows PE文件头相关的知识，以及base64和DES算法，这个exe正如题目名称，不能直接运行，用exeinfo也只能得到下图：

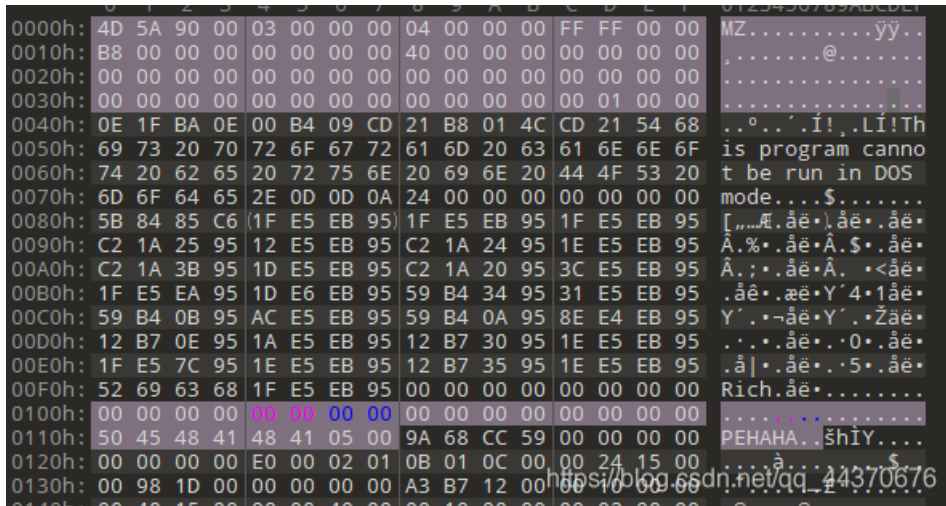


关于windows PE文件头，可以参考这篇[PE总结 - DOS文件头、PE文件头、节表和表详解](#)

关于DES算法，可以参考这位大佬的博客[DES加密与解密原理及C++代码实现](#)

预处理

预处理的主要目的是让exe文件能被IDA反编译以及可以执行,这里用010 Editor处理,打开后看到



DOS ME头 IMAGE_DOS_HEADER

定义如下：

```

IMAGE_DOS_HEADER STRUCT
+00h WORD e_magic // Magic DOS signature MZ(4Dh 5Ah) DOS可执行文件标记
+02h WORD e_cblp // Bytes on Last page of file
+04h WORD e_cp // Pages in file
+06h WORD e_crlc // Relocations
+08h WORD e_cparhdr // Size of header in paragraphs
+0ah WORD e_minalloc // Minimun extra paragraphs needs
+0ch WORD e_maxalloc // Maximun extra paragraphs needs
+0eh WORD e_ss // intial(relative)SS value DOS代码的初始化堆栈SS
+10h WORD e_sp // intial SP value DOS代码的初始化堆栈指针SP
+12h WORD e_csum // Checksum
+14h WORD e_ip // intial IP value DOS代码的初始化指令入口[指针IP]
+16h WORD e_cs // intial(relative)CS value DOS代码的初始堆栈入口 CS
+18h WORD e_lfarlc // File Address of relocation table
+1ah WORD e_ovno // Overlay number
+1ch WORD e_res[4] // Reserved words
+24h WORD e_oemid // OEM identifier(for e_oeminfo)
+26h WORD e_oeminfo // OEM information;e_oemid specific
+29h WORD e_res2[10] // Reserved words
+3ch LONG e_lfanew // Offset to start of PE header 指向PE文件头
IMAGE_DOS_HEADER ENDS

```

这里需要注意的是最后一个成员变量e_lfanew的值,它指向PE文件头,PE文件头定义放到下面,它的前4byte值为0x00004550,而此刻e_lfanew为0x0100,0x0110处的值为0x41484550,猜测e_lfanew和PE文件头值都不对,把e_lfanew修改为0x0110,0x0110处修改为0x0004550

PE文件头

PE文件头是由IMAGE_NT_HEADERS结构定义的:

```

IMAGE_NT_HEADERS STRUCT
+0h DWORD Signature PE文件标识
+4h IMAGE_FILE_HEADER FileHeader
+18h IMAGE_OPTIONAL_HEADER32 OptionalHeader
IMAGE_NT_HEADERS ENDS

```

PE文件标识即0x00004550,而FileHeader由结构体IMAGE_FILE_HEADER定义

```

IMAGE_FILE_HEADER STRUCT
+04h WORD Machine; // 运行平台
+06h WORD NumberOfSections; // 文件的区块数目
+08h DWORD TimeDateStamp; // 文件创建日期和时间
+0Ch DWORD PointerToSymbolTable; // 指向符号表(主要用于调试)
+10h DWORD NumberOfSymbols; // 符号表中符号个数(同上)
+14h WORD SizeOfOptionalHeader; // IMAGE_OPTIONAL_HEADER32 结构大小
+16h WORD Characteristics; // 文件属性
IMAGE_FILE_HEADER ENDS

```

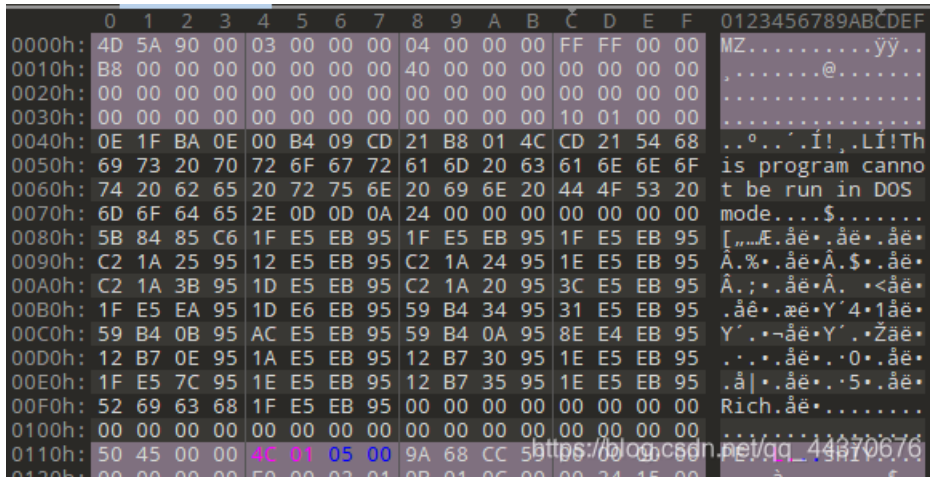
与这道题相关的是Machine的值，Machine取值有3中情况：

1.I386: 0x014C

2.IA64: 0x0200

3.AMD64: 0x8664

这里应当去第1个，那现在把0x114处修改为0x014C,这3处修改完后如下：



现在修改完后文件还是不可执行，应该是可

执行属性关闭了，修改下。

| Name | Value | Start | Size | Color | Comment |
|--|---------|-------|------|---------|-----------------------------------|
| > struct IMAGE_NT_HEADERS NtHeader | | 110h | F8h | Fg: Bg: | |
| ▼ struct IMAGE_SECTION_HEADER SectionHeaders[5] | | 208h | C8h | Fg: Bg: | |
| ▼ struct IMAGE_SECTION_HEADER SectionHeaders[0] | .text | 208h | 28h | Fg: Bg: | |
| > BYTE Name[8] | .text | 208h | 8h | Fg: Bg: | can end without zero |
| > union Misc | | 210h | 4h | Fg: Bg: | |
| DWORD VirtualAddress | 1000h | 214h | 4h | Fg: Bg: | |
| DWORD SizeOfRawData | 152400h | 218h | 4h | Fg: Bg: | |
| DWORD PointerToRawData | 400h | 21Ch | 4h | Fg: Bg: | |
| DWORD PointerToRelocations | 0h | 220h | 4h | Fg: Bg: | |
| DWORD PointerToLinenumbers | 0 | 224h | 4h | Fg: Bg: | |
| WORD NumberOfRelocations | 0 | 228h | 2h | Fg: Bg: | |
| WORD NumberOfLinenumbers | 0 | 22Ah | 2h | Fg: Bg: | |
| ▼ struct SECTION_CHARACTERISTICS Characteristics | | 22Ch | 4h | Fg: Bg: | https://blog.csdn.net/qq_44370676 |
| ULONG IMAGE_SCN_TYPE_NO_PAD : 1 | 0 | 22Ch | 4h | Fg: Bg: | 0x00000008 Reserved |

| | | | | | |
|-------------------------------------|---|------|----|---------|-------------------------------------|
| ULONG IMAGE_SCN_LNK_COMDAT : 1 | 0 | 22Ch | 4h | Fg: Bg: | 0x00001000 Section contents co... |
| ULONG IMAGE_SCN_GPREL : 1 | 0 | 22Ch | 4h | Fg: Bg: | 0x00008000 Section content can ... |
| ULONG IMAGE_SCN_MEM_16BIT : 1 | 0 | 22Ch | 4h | Fg: Bg: | 0x00020000 |
| ULONG IMAGE_SCN_MEM_LOCKED : 1 | 0 | 22Ch | 4h | Fg: Bg: | 0x00040000 |
| ULONG IMAGE_SCN_MEM_PRELOAD : 1 | 0 | 22Ch | 4h | Fg: Bg: | 0x00080000 |
| ULONG IMAGE_SCN_ALIGN_1BYTES : 1 | 0 | 22Ch | 4h | Fg: Bg: | 0x00100000 |
| ULONG IMAGE_SCN_ALIGN_2BYTES : 1 | 0 | 22Ch | 4h | Fg: Bg: | 0x00200000 |
| ULONG IMAGE_SCN_ALIGN_8BYTES : 1 | 0 | 22Ch | 4h | Fg: Bg: | 0x00400000 |
| ULONG IMAGE_SCN_ALIGN_128BYTES : 1 | 0 | 22Ch | 4h | Fg: Bg: | 0x00800000 |
| ULONG IMAGE_SCN_LNK_NRELOC_OVFL : 1 | 0 | 22Ch | 4h | Fg: Bg: | 0x01000000 Section contains exte... |
| ULONG IMAGE_SCN_MEM_DISCARDABLE : 1 | 0 | 22Ch | 4h | Fg: Bg: | 0x02000000 Section can be disca... |
| ULONG IMAGE_SCN_MEM_NOT_CACHED : 1 | 0 | 22Ch | 4h | Fg: Bg: | 0x04000000 Section is not cachable |
| ULONG IMAGE_SCN_MEM_NOT_PAGED : 1 | 0 | 22Ch | 4h | Fg: Bg: | 0x08000000 Section is not pagea... |
| ULONG IMAGE_SCN_MEM_SHARED : 1 | 0 | 22Ch | 4h | Fg: Bg: | 0x10000000 Section is shareable |
| ULONG IMAGE_SCN_MEM_EXECUTE : 1 | 1 | 22Ch | 4h | Fg: Bg: | 0x20000000 Section is executable |
| ULONG IMAGE_SCN_MEM_READ : 1 | 1 | 22Ch | 4h | Fg: Bg: | 0x40000000 Section is readable |
| ULONG IMAGE_SCN_MEM_WRITE : 1 | 0 | 22Ch | 4h | Fg: Bg: | 0x80000000 Section is writable |

从二进制方面看，这次修改把0x22C处的值从0x40000020变成了0x60000020

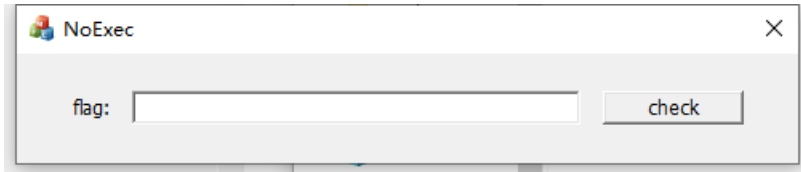
```
0210h: DF 23 15 00 00 10 00 00 00 24 15 00 00 04 00 00 B#.....$.@
0220h: 00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 40 .....@
0230h: 25 72 64 61 74 61 00 00 FA D1 04 00 00 40 15 00 rdata ün @
```

```
0210h: DF 23 15 00 00 10 00 00 00 24 15 00 00 04 00 00 B#.....$.@
0220h: 00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 60 .....@
```

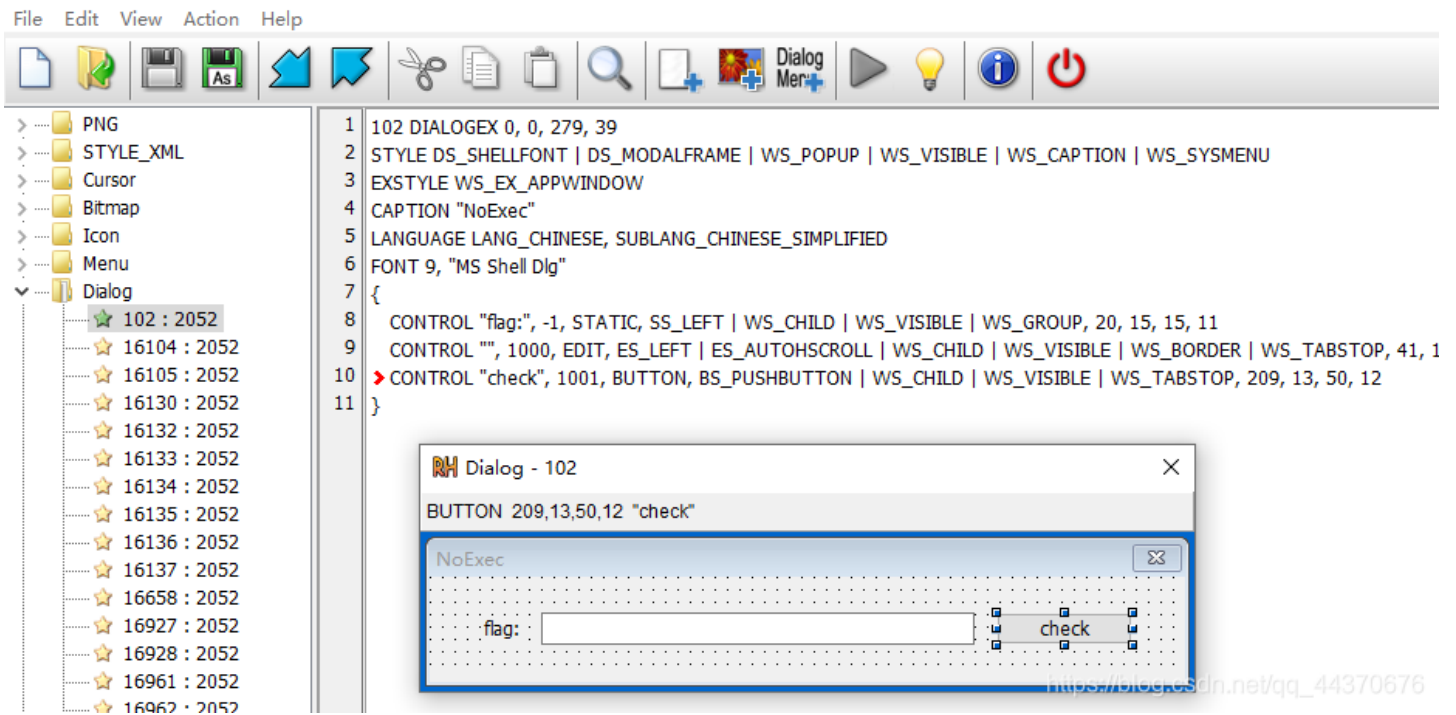
现在修改算是完成了，exe终于可以执行了

找到入口

运行一下：

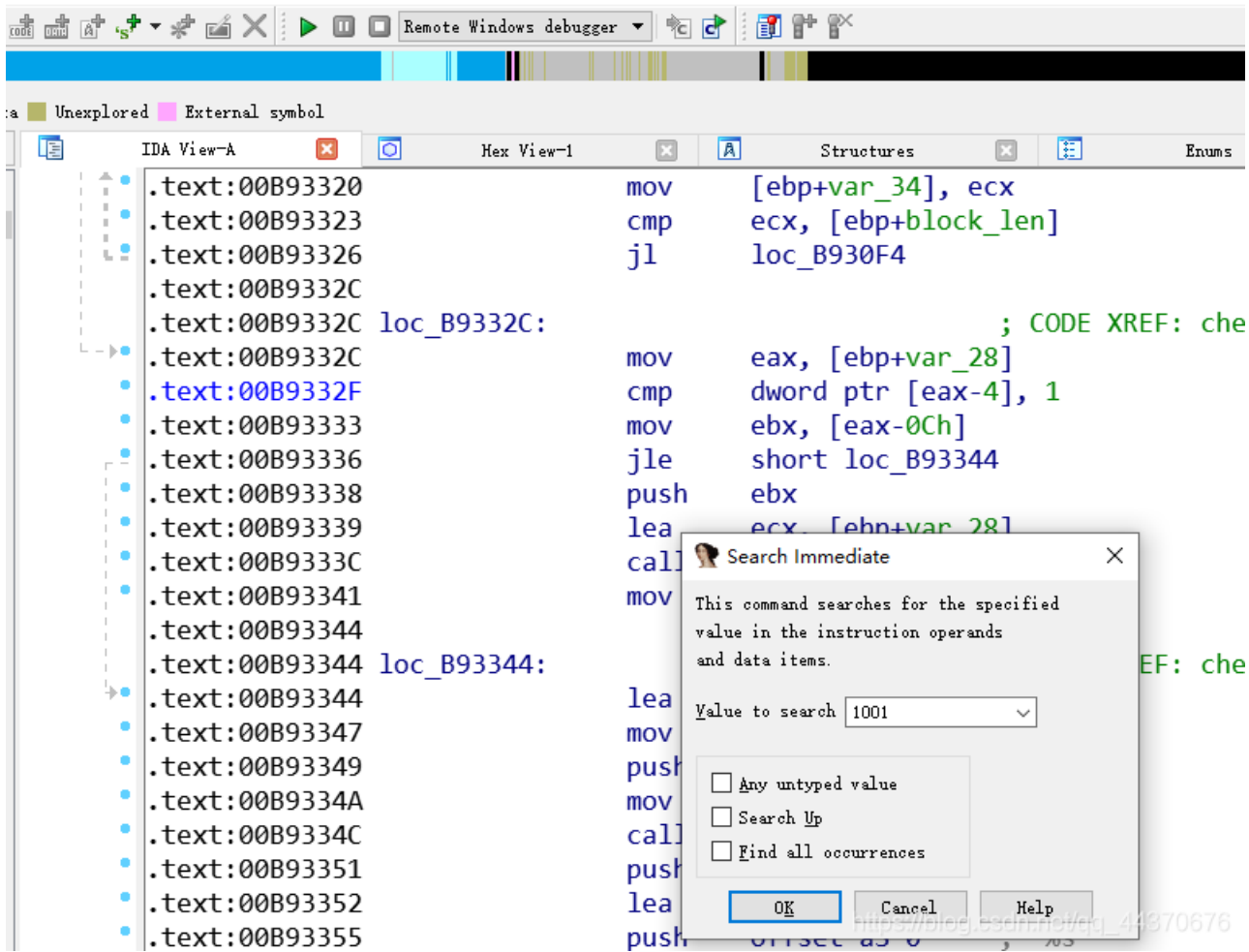


用Resource Hacker查找一下check所在的函数



check按钮的id为1001，开始IDA分析

进入IDA，搜索立即数1001



点进sub_B92C30, F5


```

15 v5 = ((int (__thiscall *) (int (__stdcall ***) (int, int))) (*v2)[3]) (v2) + 16;
16 v6 = 0;
17 sub_BA8ED5(v1, 1000, (int)&v5);
18 if ( *(_DWORD *) (v5 - 12) == 37 )
19 {
20     if ( check(&v5) == (char *)1 )
21         sub_BA254B((int)&unk_D13ECC, 0, 0);
22     else
23         sub_BA254B((int)&unk_D13ED0, 0, 0);
24 }
25 v6 = -1;
26 v3 = v5 - 16;
27 result = (volatile signed __int32 *) (v5 - 16 + 12);
28 if ( _InterlockedDecrement(result) <= 0 )
29     result = (volatile signed __int32 *) ((int (__stdcall ***) (int)) (**(_DWORD **) v3 + 4));
30 return result;

```

check函数是我重命名过的，动态调试发现，v5是输入字符串，(DWORD*)(v5 - 12)为输入字符串长度，表示输入37个字符，check函数返回1表示成功。这里动态调试，统一用字符串 `abcdefghijklmnopqrstuvwxy0123456789A`

分析check函数

check函数400多行代码，其中前面很长一部分和最后一部分都是无关紧要的，从120多行还是进入正题，部分变量重命名过。

```

128 LOBYTE(v79) = 1;
129 input_len = *(_DWORD *) (*input - 12);
130 if ( input_len > 0 )
131 {
132     do
133     {
134         v66 = (volatile signed __int32 *) (i + 1);
135         if ( (i + 1) % 2 ) // 偶数位
136         {
137             if ( i < 0 ) // 有效索引
138                 goto LABEL_35;
139             if ( i > input_len )
140                 goto LABEL_35;
141             v69 = *(_BYTE *) (*input + i);
142             v68 = *(_DWORD *) even - 3;
143             v18 = v68 + 1;
144             v70 = v68 + 1;
145             if ( v68 + 1 < 0 )
146                 goto LABEL_35;
147             if ( ((1 - *(_DWORD *) even - 1) | (*(_DWORD *) even - 2) - v18) < 0 )
148             {
149                 sub_B92750(v18);
150                 even = v77;
151                 v18 = v70;

```

字符串按索引奇偶分成2个字符串

```

139     if ( i > input_len )
140         goto LABEL_35;
141     v69 = *((_BYTE *) (*input + i));
142     v68 = *((_DWORD *) even - 3);
143     v18 = v68 + 1;
144     v70 = v68 + 1;
145     if ( v68 + 1 < 0 )
146         goto LABEL_35;
147     if ( ((1 - *((_DWORD *) even - 1)) | (*((_DWORD *) even - 2) - v18)) < 0 )
148     {
149         sub_B92750(v18);
150         even = v77;
151         v18 = v70;
152     }
153     even[v68] = v69;
154     if ( v18 > *((_DWORD *) even - 2) )
155         goto LABEL_35;
156     *((_DWORD *) even - 3) = v18;
157     even[v18] = 0;
158 }

```

https://blog.csdn.net/qq_44370676

```

text 0i 159     }
text 0i 160     else // 奇数位
text 0i 161     {
text 0i 162         if ( i < 0 )
text 0i 163             goto LABEL_35;
text 0i 164         if ( i > input_len )
text 0i 165             goto LABEL_35;
text 0i 166         v69 = *((_BYTE *) (*input + i));
text 0i 167         v68 = *((_DWORD *) odd - 3);
text 0i 168         v17 = v68 + 1;
text 0i 169         v70 = v68 + 1;
text 0i 170         if ( v68 + 1 < 0 )
text 0i 171             goto LABEL_35;
text 0i 172         if ( ((1 - *((_DWORD *) odd - 1)) | (*((_DWORD *) odd - 2) - v17)) < 0 )
text 0i 173         {
text 0i 174             sub_B92750(v17);
text 0i 175             odd = v74;
text 0i 176             v17 = v70;
text 0i 177         }
text 0i 178         odd[v68] = v69;
text 0i 179         if ( v17 > *((_DWORD *) odd - 2) )

```

https://blog.csdn.net/qq_44370676

这么一大串代码其实就是把字符串按索引(从0开始)奇偶性拆分成2个字符串，分别保存在even和odd数组中，相当于：

```

even = ''
odd = ''
input_list = [s for s in input]
for i in range(0, len(input_list), 2):
    even += input_list[i]
    odd += input_list[i + 1]

```

将2个字符串base64编码后拼接


```

197     v19 = v70;
198 }
199 v20 = (unsigned int)base64(v19, even, &v70);
200 my_sprintf((int)&Str, "%s", v20);
201 sub_B93A3d(Str, *((_DWORD *)Str - 3));
202 v21 = *((_DWORD *)odd - 3);
203 v70 = v21;
204 if ( v21 < 0 )
205     sub_B92800(-2147024809);
206 if ( ((1 - *((_DWORD *)odd - 1)) | (*((_DWORD *)odd - 2) - v21)) < 0 )
207 {
208     sub_B92750(v21);
209     odd = v74;
210     v21 = v70;
211 }
212 v22 = (unsigned int)base64(v21, odd, &v70);
213 my_sprintf((int)&Str, "%s", v22);
214 sub_B93A30(Str, *((_DWORD *)Str - 3));
215 v23 = v76;
216 if ( *((_DWORD *)v76 - 12) % 8 )
217 {
218     my_sprintf((int)&v71, "%c", 204);
219     v24 = *((_DWORD *)v23 - 12) % 8;
220     v70 = 8 - v24;
221     if ( 8 - v24 > 0 )
222     {
223         v25 = v71;
224         v26 = 8 - v24;
225         do
226         {
227             sub_B93A30(v25, *((_DWORD *)v25 - 3));
228             --v26;
229         }
230         while ( v26 );
231         v11 = v78;
232         even = v77;
233         v23 = v76;
234     }
235 }

```

// v23,v76均为base64拼接后的字符串
// base64拼接后如果字符串长度不为8的倍数，那么将\xCC填充到base64后的字符串，输入37位，拆分再base64

https://blog.csdn.net/qq_44370676

这里部分函数名也重命名过，借助IDA的findcrypt插件发现有base64_box调用，这个base64函数第2个参数为待编码字符串，返回值是编码后的字符串。这一段将上面拆分过的字符串分别base64编码，再拼接到一起，放到str里，通过动态调试发现，拼接后的字符串不知怎么的传到v23和v76中。

输入 `abcdefghijklmnopqrstuvwxyz0123456789A`,处理后应该为 `YWNlZ2Z1rbW9xc3V3eTAyNDY4QQ==YmRmaGpsbnBydHZ4ejEzNTc5`

```

xt 01 215 v23 = v76;
xt 01 216 if ( *((_DWORD *)v76 - 12) % 8 )
xt 01 217 {
xt 01 218     my_sprintf((int)&v71, "%c", 204);
xt 01 219     v24 = *((_DWORD *)v23 - 12) % 8;
xt 01 220     v70 = 8 - v24;
xt 01 221     if ( 8 - v24 > 0 )
xt 01 222     {
xt 01 223         v25 = v71;
xt 01 224         v26 = 8 - v24;
xt 01 225         do
xt 01 226         {
xt 01 227             sub_B93A30(v25, *((_DWORD *)v25 - 3));
xt 01 228             --v26;
xt 01 229         }
xt 01 230         while ( v26 );
xt 01 231         v11 = v78;
xt 01 232         even = v77;
xt 01 233         v23 = v76;
xt 01 234     }
xt 01 235 }

```

// v23,v76均为base64拼接后的字符串
// base64拼接后如果字符串长度不为8的倍数，那么将\xCC填充到base64后的字符串，输入37位，拆分再base64

https://blog.csdn.net/qq_44370676

这个if循环就做一件事，就是如果拼接后的字符串长度不会8的倍数，就填充\xCC到8的倍数。

输入37个字符，拆分成even长度19, base64后长度为28，拆分成的odd长度18, base64后长度24.相加长度52，需要补充4个\xCC。

即 `YWNlZ2Z1rbW9xc3V3eTAyNDY4QQ==YmRmaGpsbnBydHZ4ejEzNTc5\xcc\xcc\xcc\xcc`

将拼接后的字符串 DES加密

```

t 01 236 sub_B934E0((int)&v72, v61, v62);
t 01 237 block_index = 0;
t 01 238 base_length = *((_DWORD *)v23 - 12);
t 01 239 v70 = 0;

```

```

t 001 240 block_len = base_length / 8;
t 001 241 if ( base_length / 8 > 0 )
t 001 242 {
t 001 243     v29 = v72;
t 001 244     do
t 001 245     {
t 001 246         v30 = (void **)sub_B935D0(&v76, (int)&v63, 8 * block_index, 8);
t 001 247         LOBYTE(v79) = 8;
t 001 248         v31 = (char *)*v30;
t 001 249         v32 = (int)(v11 - 16);
t 001 250         Src = v31;
t 001 251         v66 = (volatile signed __int32 *)(v11 - 16);
t 001 252         if ( v31 - 16 != v11 - 16 )
t 001 253         {
t 001 254             v33 = *(_DWORD *)(v32 + 12) < 0;
t 001 255             v67 = (volatile signed __int32 *)(v32 + 12);
t 001 256             even = v77;

```

这里开始将字符串8个一组处理，很有可能是DES（AES 16个一组）

```

001 350     v67 = (volatile signed __int32 *)v11;
001 351     if ( !((unsigned __int8)(v33 ^ v44) | v39) )
001 352     {
001 353         sub_B92680(*(_DWORD *)(v29 - 12));
001 354         v29 = v72;
001 355     }
001 356     if ( *(_DWORD *)v11 - 1 > 1 )
001 357     {
001 358         sub_B92680(*(_DWORD *)v11 - 3);
001 359         v11 = v78;
001 360     }
001 361     sub_B91EC0(v29, (int)v11, v67);
001 362     sub_B93A30(v11, *(_DWORD *)v11 - 3);
001 363     block_index = v70 + 1;
001 364     v70 = block_index;
001 365 }
001 366 while ( block_index < block_len );
001 367 }
001 368 v45 = v73;
001 369 v46 = *(_DWORD *)v73 - 3;

```

这个do while里面 v11出镜率比较大，调试看下

大，调试看下

```

05DEF8 db 59h ; Y
05DEF9 db 57h ; W
05DEFA db 4Eh ; N
05DEFB db 6Ch ; l
05DEFC db 5Ah ; Z
05DEFD db 32h ; 2
05DEFE db 6Ch ; l
05DEFF db 72h ; r

```

也就拼接字符串前8个字符，v29的值为

```

debug016:0105E000 db 6Eh ; r|
debug016:0105E001 db 6
debug016:0105E002 db 15h

```

```
• debug016:0105E003 db 51h ; Q
• debug016:0105E004 db 93h
• debug016:0105E005 db 5Bh ; [
• debug016:0105E006 db 7
• debug016:0105E007 db 0EAh
```

点进sub_B91EC0看看

```
U 86 v3 = edx0;
O 87 v62 = a1;
O 88 v4 = a2;
O 89 v5 = malloc(8u);
O 90 v6 = v5;
O 91 v64 = v5;
O 92 v61 = (int)(v5 + 4);
O 93 des_process((unsigned __int8 *)&IP_table_post_half, v4, v5, 4u);
O 94 des_process((unsigned __int8 *)&IP_table_pre_half, v4, v6 + 4, 4u);
O 95 des_process((unsigned __int8 *)&PC_1, v3, keyPCI1, 7u);
O 96 v65 = 8;
O 97 v59 = (char *)(v77 - v6);
O 98 v7 = (char *)&left;
O 99 v60 = (char *)(v77 - (v6 + 4));
O 100 v63 = (char *)&left;
O 101 do
O 102 {
```

https://blog.csdn.net/qq_44370676

这个IP_table变量名和des_process都是手动改过的，查看全局变量值很容易发现这些des_box，不过findcrypt竟然没查出来！！！！（气死）。所以目前可以大致认定这是DES算法了，现在得确定密钥值，v3是传入的第一个参数，就是之前的v29，而用到PC1交换表的步骤就是密钥变换，所以认定v29就是密钥，经过多次调试发现，这是个常量。所以DES用到的密钥为 `\x6e\x06\x15\x51\x93\x5b\x07\xea`。找不到IV向量，就暂且认定是ECB模式。

```

360     }
361     sub_111EC0(v29, (int)v11, v67);
362     sub_113A30(v11, *((_DWORD *)v11 - 3));
363     block_index = v70 + 1;
364     v70 = block_index;
365     }
366     while ( block_index < block_len );
367 }
368 v45 = v73;
369 v46 = *((_DWORD *)v73 - 3);
370 if ( *((_DWORD *)v73 - 1) > 1 )
371 {
372     sub_112680(*((_DWORD *)v73 - 3));
373     v45 = v73;
374 }
375 v47 = (unsigned int)base64(v46, v45, &v70);
376 my_sprintf((int)&Str, "%s", v47);
377 v48 = Str;
378 index = 0;
379 if ( *((_DWORD *)Str - 3) <= 0 )

```

v45 ,v73经过调试发现就是DES加密后的

值，为

```

debug015:006BE00F db 0
debug015:006BE010 db 17h
debug015:006BE011 db 21h ; !
debug015:006BE012 db 0Bh
debug015:006BE013 db 18h
debug015:006BE014 db 50h ; P
debug015:006BE015 db 65h ; e
debug015:006BE016 db 0B4h
debug015:006BE017 db 22h ; "
debug015:006BE018 db 5Bh ; [
debug015:006BE019 db 38h ; 8
debug015:006BE01A db 7Bh ; {
debug015:006BE01B db 9
debug015:006BE01C db 41h ; A
debug015:006BE01D db 0C6h
debug015:006BE01E db 0A8h
debug015:006BE01F db 0EEh
debug015:006BE020 db 28h ; (
debug015:006BE021 db 54h ; T
debug015:006BE022 db 0B2h
debug015:006BE023 db 26h ; &
debug015:006BE024 db 0FDh
debug015:006BE025 db 0E3h
debug015:006BE026 db 1Dh

```

运行以下脚本对比一下：

```

import base64
from Crypto.Cipher import DES

def encrypt():
    input = 'abcdefghijklmnopqrstuvwxyz0123456789A'
    str1 = ''
    str2 = ''

    input_list = [s for s in input]
    for i in range(0, len(input_list), 2):
        str1 += input_list[i]
        if i + 1 < len(input_list):
            str2 += input_list[i + 1]

    b64_str = (base64.b64encode(str1.encode('utf-8')) + base64.b64encode(str2.encode('utf-8'))) + b'\xcc\xcc\xcc\xcc'

    key = b'\x6e\x06\x15\x51\x93\x5b\x07\xea'
    des = DES.new(key, DES.MODE_ECB)
    des_encrypt_str = des.encrypt(b64_str)

    print(des_encrypt_str)

if __name__ == '__main__':
    encrypt()

```

结果为

```

b'\x17!\x0b\x18Pe\xb4"[8{\t\xa\xc6\xa8\xee(T\xb2&\xfd\xe3\x1dq\xfaI\xd7\xbe\x14\xe6s\xb0B$\x98\x14\xaac7\xdb\xc2+I\xe5R\xe2\x177X)Y\xbf\xb3`q\x07'

```

可以得知这里DES采用的是ECB模式

再一次base64

```

369 v46 = *((_DWORD *)v73 - 3);
370 if ( *((_DWORD *)v73 - 1) > 1 )
371 {
372     sub_112680(*((_DWORD *)v73 - 3));
373     v45 = v73;
374 }
375 v47 = (unsigned int)base64(v46, v45, &v70);
376 my_sprintf((int)&Str, "%s", v47);
377 v48 = Str;
378 index = 0;
379 if ( *((_DWORD *)Str - 3) <= 0 )
380 {
381 LABEL_94:
382     v70 = 1;
383     |
384     else
385     {
386         v50 = Str - "GcDk0SvnNA1tsmp5FCK1FpSDfUXZbhHBSPhZaixuMyzqyysOAPCPB/p7sMpmK1KZo+1PfhMZxw=";
387         block_len = Str - "GcDk0SvnNA1tsmp5FCK1FpSDfUXZbhHBSPhZaixuMyzqyysOAPCPB/p7sMpmK1KZo+1PfhMZxw=";
388         while ( 1 )
389         {
390             if ( index < 0 )
391                 goto LABEL_35;

```

https://blog.csdn.net/qq_44370576

这里把DES过的字符串再一次base64编码保存到Str中，拿str与常量字符

串 GcDk0SvnNA1tsmp5FCK1FpSDfUXZbhHBSPhZaixuMyzqyysOAPCPB/p7sMpmK1KZo+1PfhMZxw= 对比。

```

386 v50 = Str - "GcDk0SvnNA1tsmp5FCK1FpSDfUXZbhHBSPheZaixuMyzqyysOAPCPB/p7sMpmK1KZo+1PfhMZxw=";
387 block_len = Str - "GcDk0SvnNA1tsmp5FCK1FpSDfUXZbhHBSPheZaixuMyzqyysOAPCPB/p7sMpmK1KZo+1PfhMZxw=";
388 while ( 1 )
389 {
390     if ( index < 0 )
391         goto LABEL_35;
392     last_len = *((_DWORD *)Str - 3);
393     if ( index > last_len )
394         goto LABEL_35;
395     if ( aGcdk0svnna1tsm[index + v50] != aGcdk0svnna1tsm[index] )
396         break;
397     v50 = block_len;
398     if ( ++index >= last_len )
399         goto LABEL_94;
400 }
401 v70 = -1;
402 }

```

https://blog.csdn.net/qq_44370676

这个比对过程很有意思，大概时goto LABEL_35最终会失败， goto LABEL_94会成功，LABEL_94上面的if循环应该是永远不会成立，只能这样执行。而v50必须为0。所以确

定 `GcDk0SvnNA1tsmp5FCK1FpSDfUXZbhHBSPheZaixuMyzqyysOAPCPB/p7sMpmK1KZo+1PfhMZxw=` 是最终结果。

二.解题脚本

```

import base64
from Crypto.Cipher import DES

def decrypt():
    code = "GcDk0SvnNA1tsmp5FCK1FpSDfUXZbhHBSPheZaixuMyzqyysOAPCPB/p7sMpmK1KZo+1PfhMZxw=".encode('utf-8')
    des_encrypt_str = base64.b64decode(code)

    key = b'\x6e\x06\x15\x51\x93\x5b\x07\xea'
    des = DES.new(key, DES.MODE_ECB)

    b64_str = des.decrypt(des_encrypt_str)

    str1 = b64_str[:28]
    str2 = b64_str[28:]

    even = base64.b64decode(str1)
    odd = base64.b64decode(str2)

    flag = ''
    for i in range(18):
        flag += chr(even[i]) + chr(odd[i])
    flag += chr(even[18])
    print(flag)

if __name__ == '__main__':
    decrypt()

```

得到flag: `flag{PE_StrUcTuRe_1s_Very_Imp0rtant!}`

三.总结

关于这道题涉及到的PE文件头和DES算法都该好好学习下，目前还只是学到了些皮毛，以后会再接再厉，欢迎大佬们来指教。