CTF小白学习笔记(Reverse)-i春秋 NoExec

原创

I still … 于 2020-12-10 15:37:43 发布 506 收藏 2

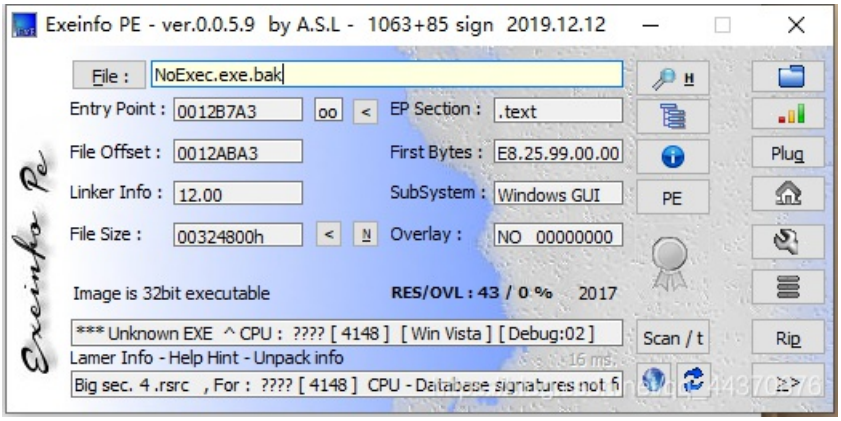
分类专栏： CTF的writeup 文章标签： 安全 信息安全

本文链接：https://blog.csdn.net/qq_44370676/article/details/110949986

大纲

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：

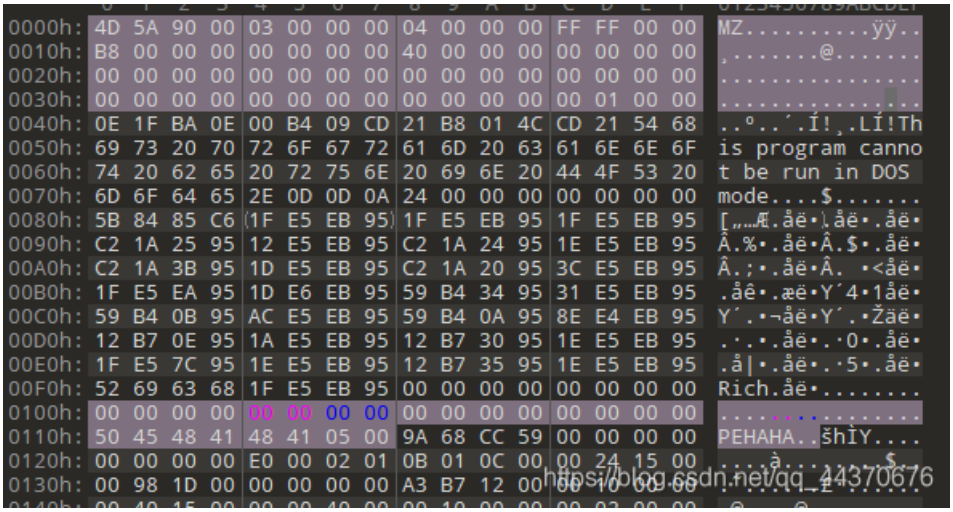


关于windows PE文件头，可以参考这篇PE总结 - DOS文件头、PE文件头、节表和表详解

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
```

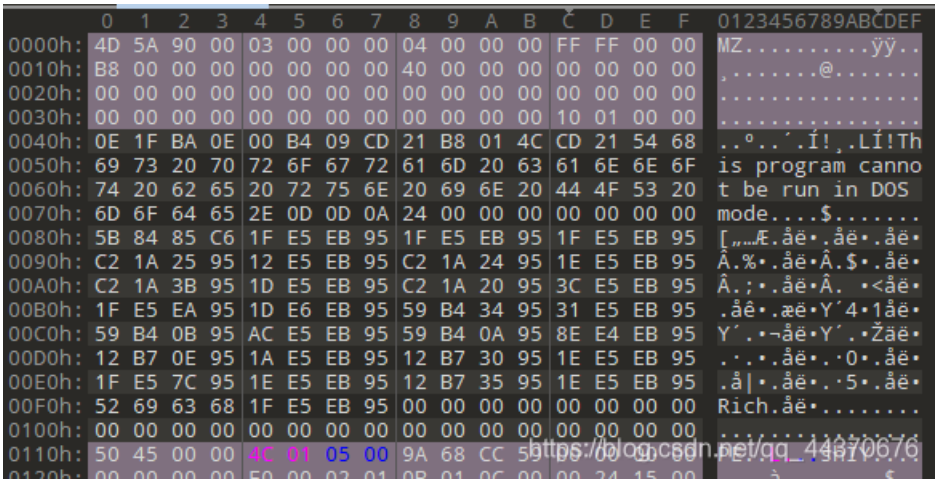
与这道题相关的是Machine的值，Machine取值有3中情况：

1.I386: 0x014C

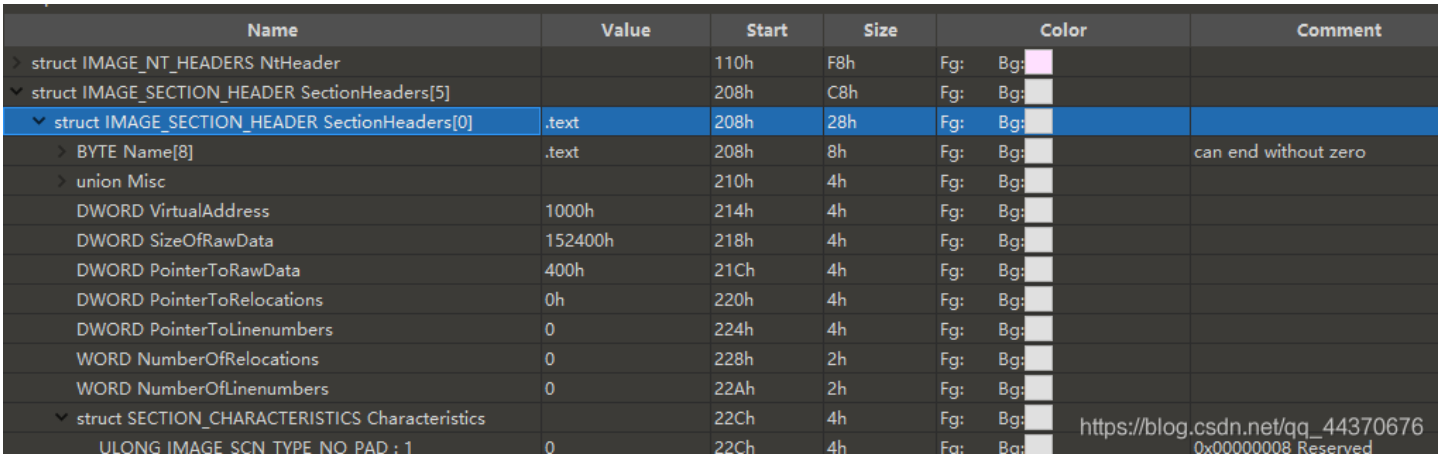
2.IA64: 0x0200

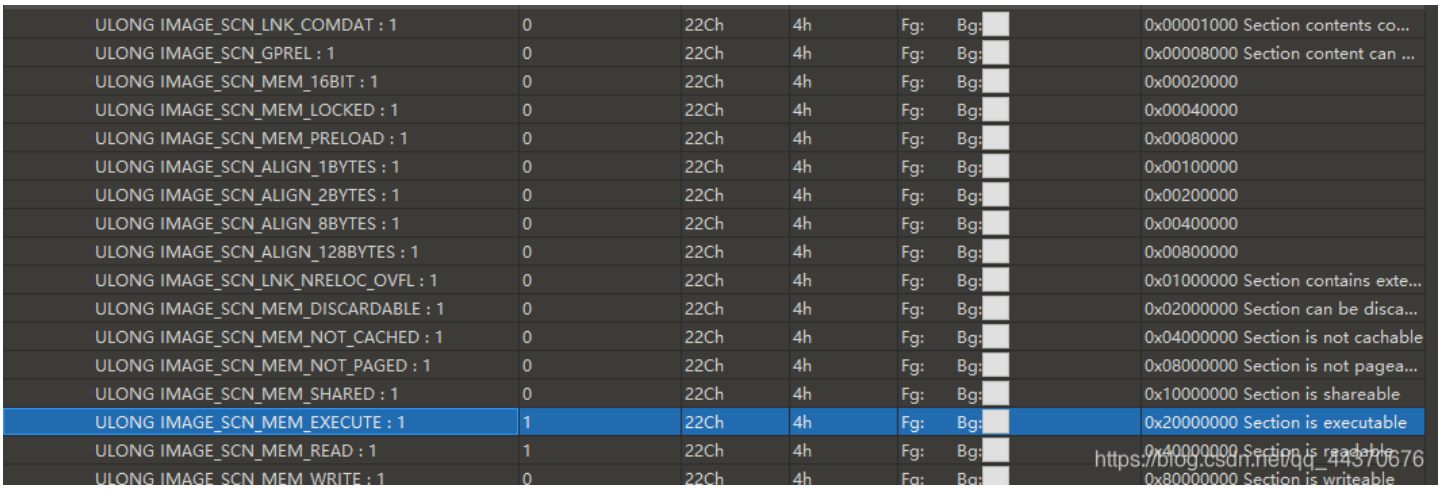
3.AMD64: 0x8664

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：

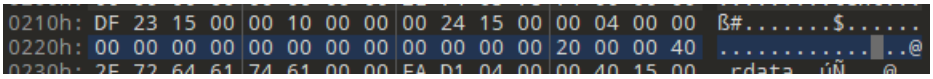


现在修改完后文件还是不可执行，应该是可执行属性关闭了，修改下。





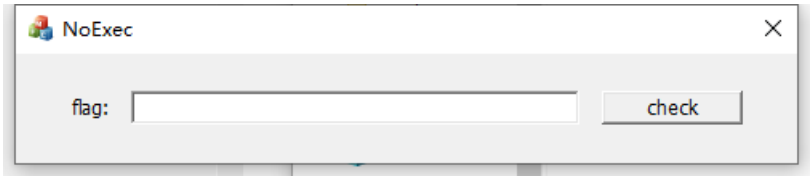
从二进制方面看，这次修改把0x22C处的值从0x40000020变成了0x60000020



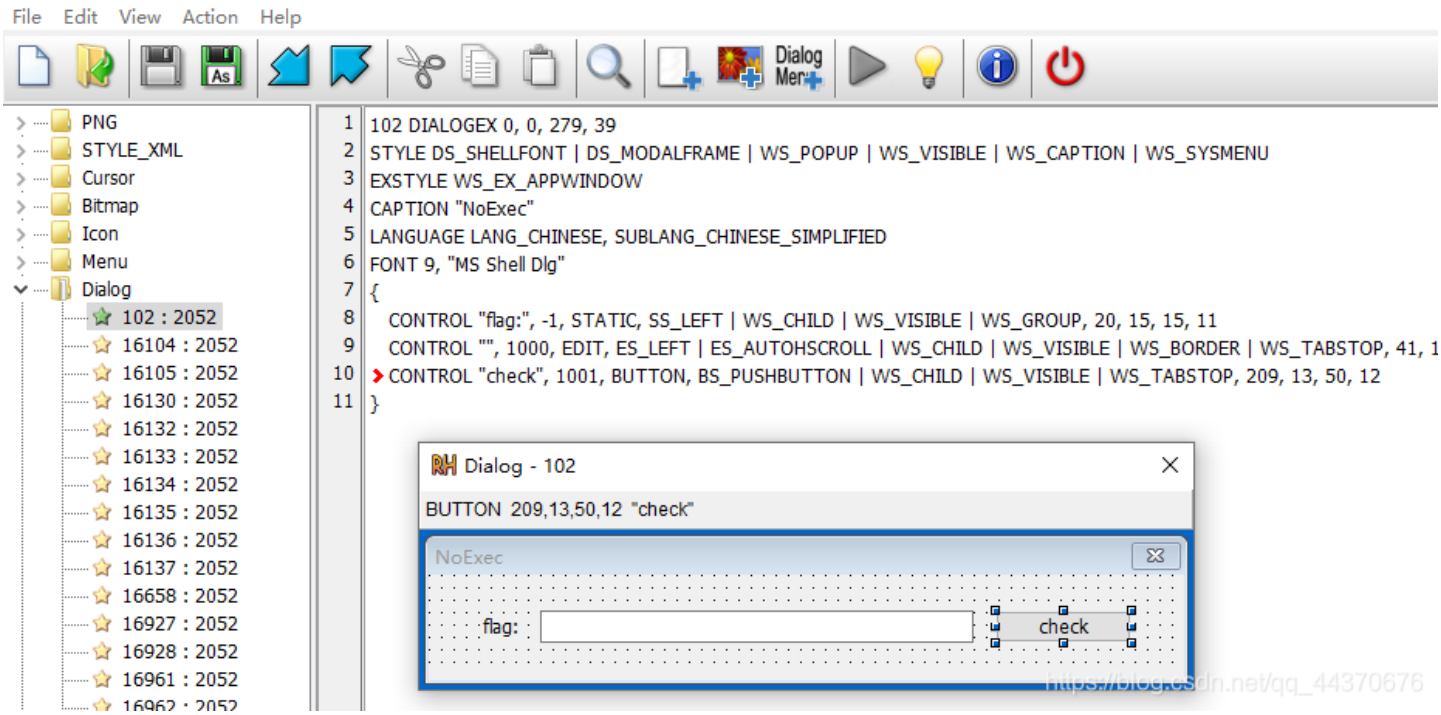
现在修改算是完成了，exe终于可以执行了

找到入口

运行一下：

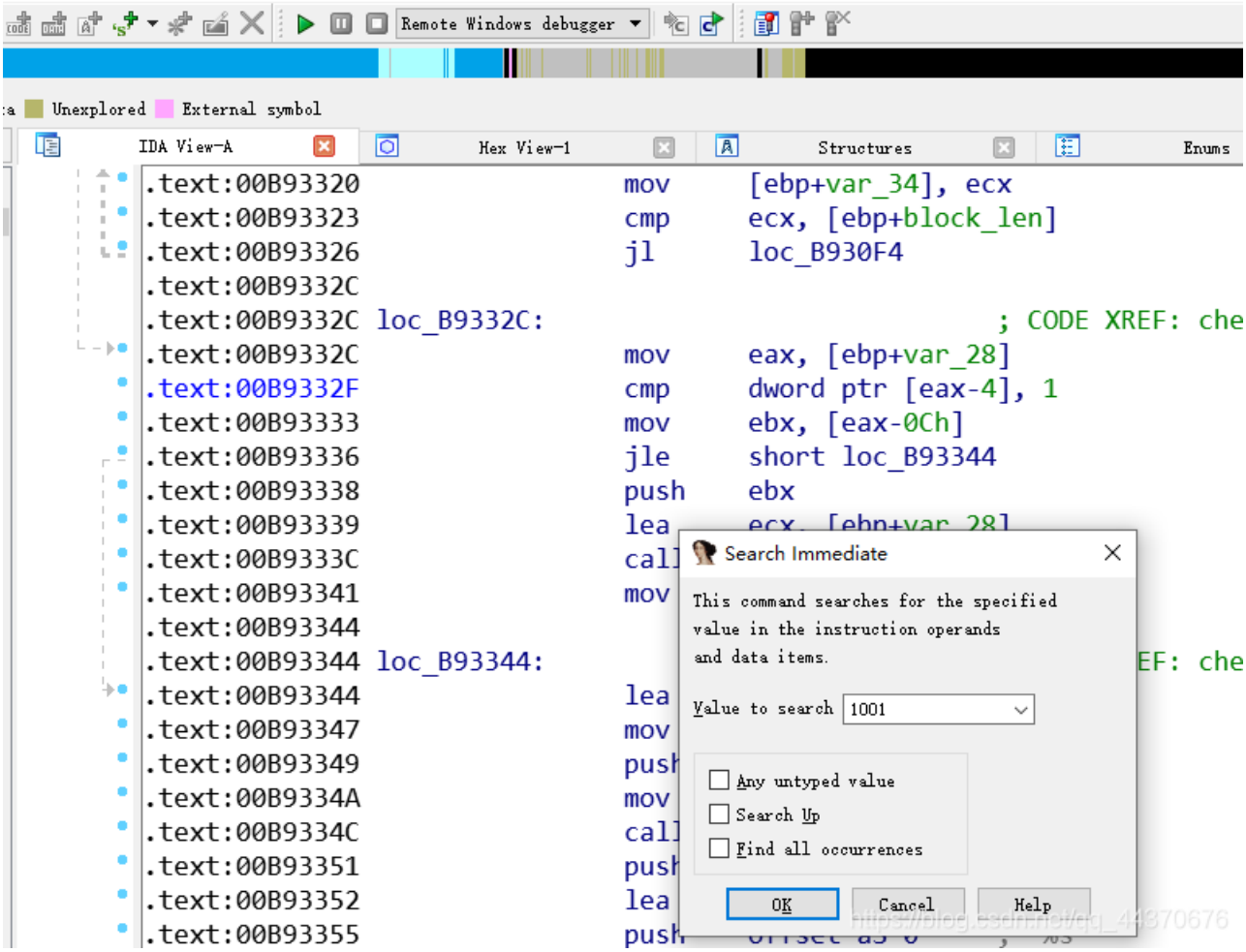


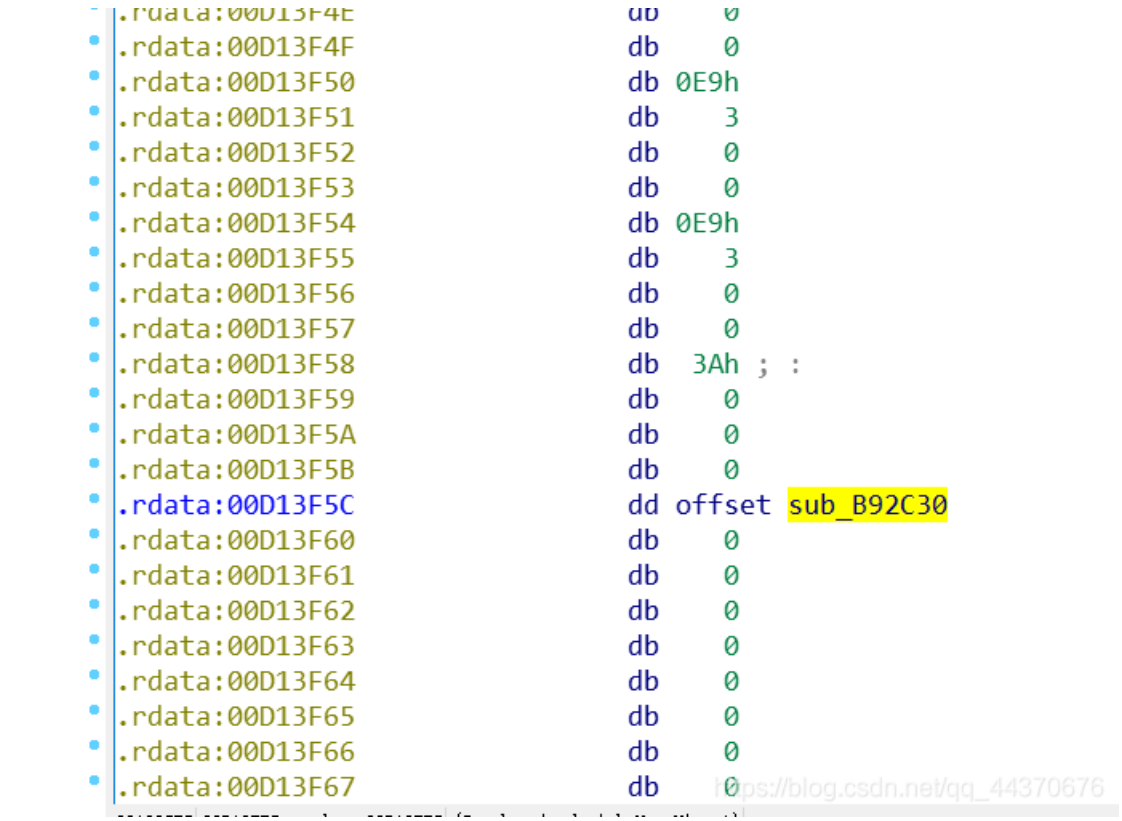
用Resource Hacker查找一下check所在的函数



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 `abcdefghijklmnopqrstuvwxyz0123456789A`

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) = 7;
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

```
        if ( i > input_len )
          goto LABEL_35;
        v69 = *(_BYTE *)(*input + i);
        v68 = *((_DWORD *)even - 3);
        v18 = v68 + 1;
        v70 = v68 + 1;
        if ( v68 + 1 < 0 )
          goto LABEL_35;
        if ( ((1 - *((_DWORD *)even - 1)) | (*((_DWORD *)even - 2) - v18)) < 0 )
        {
          sub_B92750(v18);
          even = v77;
          v18 = v70;
        }
        even[v68] = v69;
        if ( v18 > *((_DWORD *)even - 2) )
          goto LABEL_35;
        *((_DWORD *)even - 3) = v18;
        even[v18] = 0;
      }
```



```
      else                                    // 奇数位
      {
        if ( i < 0 )
          goto LABEL_35;
        if ( i > input_len )
          goto LABEL_35;
        v69 = *(_BYTE *)(*input + i);
        v68 = *((_DWORD *)odd - 3);
        v17 = v68 + 1;
        v70 = v68 + 1;
        if ( v68 + 1 < 0 )
          goto LABEL_35;
        if ( ((1 - *((_DWORD *)odd - 1)) | (*((_DWORD *)odd - 2) - v17)) < 0 )
        {
          sub_B92750(v17);
          odd = v74;
          v17 = v70;
        }
        odd[v68] = v69;
        if ( v17 > *(( DWORD *)odd - 2) )
```


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_B93A30(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;                              // v23,v76均为base64拼接后的字符串
216 if ( *(_DWORD *)(v76 - 12) % 8 )        // 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后
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 }
```

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`YWNlZ2lrbW9xc3V3eTAyNDY4QQ==YmRmaGpsbnBydHZ4ejEzNTc5`

```
215 v23 = v76;                                  // v23,v76均为base64拼接后的
216 if ( *(_DWORD *)(v76 - 12) % 8 )            // base64拼接后如果字符串长
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 }
```

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`YWNlZ2lrbW9xc3V3eTAyNDY4QQ==YmRmaGpsbnBydHZ4ejEzNTc5\xcc\xcc\xcc\xcc`

将拼接后的字符串DES加密

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

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

```
350      v67 = (volatile signed __int32 *)v11;
351      if ( !((unsigned __int8)(v33 ^ v44) | v39) )
352      {
353        sub_B92680(*(_DWORD *)(v29 - 12));
354        v29 = v72;
355      }
356      if ( *((_DWORD *)v11 - 1) > 1 )
357      {
358        sub_B92680(*((_DWORD *)v11 - 3));
359        v11 = v78;
360      }
361      sub_B91EC0(v29, (int)v11, v67);
362      sub_B93A30(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);
```


这个do while里面 v11出镜率比较大，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 ; n
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

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


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：

```python
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{\tA\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 - "GcDk0SvnNA1tsmp5FCK1FpSDfUXZbhHBSPheZaixuMyzqyysOAPCPB/p7sMpmK1KZo+lPfhMZxw=";
387     block_len = Str - "GcDk0SvnNA1tsmp5FCK1FpSDfUXZbhHBSPheZaixuMyzqyysOAPCPB/p7sMpmK1KZo+lPfhMZxw=";
388     while ( 1 )
389     {
390       if ( index < 0 )
391         goto LABEL_35;
```


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`GcDk0SvnNA1tsmp5FCK1FpSDfUXZbhHBSPheZaixuMyzqyysOAPCPB/p7sMpmK1KZo+lPfhMZxw=`对比。

```
{
  v50 = Str - "GcDk0SvnNA1tsmp5FCK1FpSDfUXZbhHBSPheZaixuMyzqyysOAPCPB/p7sMpmK1KZo+lPfhMZxw=";
  block_len = Str - "GcDk0SvnNA1tsmp5FCK1FpSDfUXZbhHBSPheZaixuMyzqyysOAPCPB/p7sMpmK1KZo+lPfhMZxw=";
  while ( 1 )
  {
    if ( index < 0 )
      goto LABEL_35;
    last_len = *((_DWORD *)Str - 3);
    if ( index > last_len )
      goto LABEL_35;
    if ( aGcdk0svnna1tsm[index + v50] != aGcdk0svnna1tsm[index] )
      break;
    v50 = block_len;
    if ( ++index >= last_len )
      goto LABEL_94;
  }
  v70 = -1;
}
```


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+lPfhMZxw=`是最终结果。

二.解题脚本

```python
import base64
from Crypto.Cipher import DES

def decrypt():
    code = "GcDk0SvnNA1tsmp5FCK1FpSDfUXZbhHBSPheZaixuMyzqyysOAPCPB/p7sMpmK1KZo+lPfhMZxw=".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。